A Visual Node-Based Programming Editor for Educational Purposes

Benjamin Stuntner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 26, 2016

Benjamin Stuntner

Contents

Declaration **iii**

Abstract **vi**

Kurzfassung **vii**

1 Introduction **1**
1.1 Research Question 1
1.2 Structure 1

2 Visual Flow-Based Programming **3**
2.1 Visual Programming 3
2.1.1 Definition 3
2.1.2 Common VPL Systems 5
2.1.3 Advantages 10
2.1.4 Disadvantages 11
2.2 Flow-Based Programming 12
2.2.1 History 12
2.2.2 Definition 13
2.2.3 Terminology 15
2.3 State of the Art 16
2.3.1 NoFlo 16
2.3.2 Unreal Engine 4 18
2.3.3 vvvv 19

3 Prototype — NO:LE:AP **21**
3.1 Early Concepts 21
3.2 Current Version 22
3.2.1 Network 24
3.2.2 Nodes 26
3.2.3 Game Entities 31
3.2.4 Worlds and Tasks 32
3.2.5 Persistency 33
3.2.6 Metrics 35

3.3 Early Playtest Results 38

4 Implementation **41**
4.1 Client 41
4.1.1 Software Stack 42
4.1.2 Node System 43
4.1.3 Network Editor 47
4.1.4 Worlds and Tasks 49
4.1.5 Persistency and Metrics 50
4.2 Server 51
4.2.1 Software Stack 51
4.2.2 Implementation 52

5 Evaluation **55**
5.1 Results 55
5.1.1 Demographic Data 56
5.1.2 Played Worlds 57
5.2 Interpretation 58
5.2.1 Analysis of the Progression System 58
5.2.2 Supporting the Thesis's Theory 59

6 Conclusion **63**
6.1 Summary 63
6.2 Problems 63
6.3 Outlook 64

A Contents of the CD-ROM **65**
A.1 Thesis 65
A.2 Online Sources 65
A.3 Images 66
A.4 NO:LE:AP 66

References **67**
Literature 67
Online sources 68

Abstract

Visual programming languages are mostly described as a two-edged sword. Since a few centuries, this is a topic that regularly increases in interest, leading to lots of discussions and several written works , but finally starts to decline again. For a few people, visual programming is the future, making textual programming languages obsolete for the majority, but most of the studies about it are by far less positive. Nonetheless, there is a huge increase in interest during the last few years. Specialized solutions for areas where non-programmers have to work with it getting more and more attention.

This thesis addresses *visual flow-based programming languages* and describes a prototypical approach to a learning game incorporating a visual node-based programming editor for educational purposes as its core feature. There will be not only the theory and current examples to given topic presented, but also how they build the basis for the described approach, the way its concepts are developed and how they get implemented. The goal of this thesis is to find out if such an editor can help to lower the entry level to areas like programming or math. There will be an in-depth description of the implemented systems gathering valuable data and finally an evaluation of first tests and their results.

Kurzfassung

Visuelle Programmiersprachen gelten für Viele als zweischneidiges Schwert. Seit mittlerweile einigen Jahrzehnten ist es ein Thema, das immer wieder an Interesse aufnimmt, für viel Gesprächsstoff sorgt und einige Arbeiten darüber produziert werden, der ganze Rummel aber schließlich wieder abklingt. Während einige Stimmen in visuellem Programmieren die Zukunft sehen, in der textuelle Sprachen für die Allgemeinheit überflüssig werden, fallen die meisten Studien eher ernüchternd aus. Trotzdem ist gerade in den letzten Jahren das Interesse dafür drastisch angestiegen. Besonders als spezialisierte Lösungen für Bereiche, in denen Nicht-Programmierer damit produktiv arbeiten, bekommt dieses Thema sehr viel Aufmerksamkeit.

Diese Arbeit beschäftigt sich mit *flussbasierten visuellen Programmiersprachen* und beschreibt einen prototypischen Ansatz eines Lernspiels, welches als Kernelement einen visuellen, knoten-basierten Editoren für Bildungszwecke integriert hat. Es werden nicht nur die Theorie zu besagtem Themengebiet und aktuelle Beispiele präsentiert, sondern auch wie sie als Basis für den eigenen Ansatz dienen und wie dieser konzeptioniert und schließlich auch implementiert wird. Das Ziel dieser Arbeit ist es, herauszufinden, ob ein derartiger Editor helfen kann, die Einstiegshürde für Bereiche wie Programmieren oder Mathematik zu senken. Es wird außerdem nicht nur erklärt, welche Technik realisiert wurde, um analysierbare Daten zu sammeln, sondern auch die Ergebnisse erster Testläufe und was dabei herausgefunden werden konnte.

Chapter 1

Introduction

This thesis is the result of combining several major interests of the author into one scientific project. First of all, game development is a very important topic for him and multiple games where created by him successfully in the past. Thus, it was one of the main goals to somehow incorporate it into this thesis. Another one are visual tools as there where several projects in the past he participated in and because these tools are getting lots of attention nowadays with a multitude of popular software companies implementing their own solutions. Finally, a topic he is very interested in too is education, which is furthermore very fitting as the authors goal was to create something with a certain kind of serious background. Combining these areas resulted into the current title — a visual node-based programming editor for educational purposes — and a confined enough topic for this thesis.

1.1 Research Question

By taking just stated thoughts into account, the following research question was formulated:

Is a visual node-based editor capable of lowering the entry level to areas like programming or math? Can it enable users with divergent knowledge in these fields to perform roughly similar? Can such an editor thus be described as a beginner-friendly tool?

1.2 Structure

After the thesis itself and the author's reasons to choose this subject where presented in chapter 1, chapter 2 introduces the reader to the *core topics of this paper*, with an in-depth explanation of visual flow-based programming on a term-by-term basis and an overview of relevant examples.

Chapter 3 introduces the *prototypical approach of this thesis* and gives an overview over the conceptional stage early on. Furthermore, the most

important parts and systems that actually made it into the current version are explained.

In chapter 4, the focus lies on the technical aspect. It explains the *implementation of the core mechanics and systems in place*. This project includes a client as well as a server software and both will be presented separately.

Chapter 5 is an evaluation, *objectively summarizing existing data that then gets analyzed and interpreted*. This chapter will try to find out if this prototypical approach had any success in finding valuable data to support the theory of this thesis.

Chapter 6 concludes this thesis by highlighting eventual problems and giving an outlook on the future of the project.

Chapter 2

Visual Flow-Based Programming

Although this thesis is using the term *node-based*, a much broader and better known one is *flow-based* — therefore, we are talking about flow-based in this chapter. The reason to choose node-based for the thesis's title is because the visual representation in the prototype's implementation resembles nodes and lots of other projects in this field are also using this term whenever they have similar visual elements.

Visual flow-based programming is a combination of two specific and well-defined terms — *visual programming* and *flow-based programming*. This chapter gives an in-depth explanation of the terminology and a brief overview of current projects utilizing visual flow-based programming.

2.1 Visual Programming

Since the early 70s, when the first *Visual Programming Languages (VPLs)* started to appear and research began in this area, there is a downright rift dividing developers and scientists whether a VPL is a better and more effective way of developing a computer program. Since then, a range of arguments for and against VPLs emerged (see sections 2.1.3 and 2.1.4).

This section explains, what VPLs are, which types of visual systems exist and further brings up some advantages and disadvantages of them.

2.1.1 Definition

While non-visual, i.e., textual programming languages are a one-dimensional stream of characters, VPLs propose a two-dimensional (or more) computation model [3]. Although VPLs are defined as (generalized) icons from an iconic system forming iconic sentences [5], these visual environments have still some kind of logical functionality underneath, therefore visual program-

ming is still considered as programming. Thus, VPL can also be classified regarding their scope, namely *general-purpose languages* and *domain-specific languages*.

Another way to distinguish VPLs is based on the amount the system relies on the graphical representation [2, 12], with a multitude of variations from which some remarkable examples are presented in this section.

Pure VPLs

There are some criteria a VPL has to fulfill to be a *pure* visual programming system. First of all, the system is capable of executing its program. There are lots of graphical tools supporting the software development process and its multiple phases, but they neither produce real code nor execute — in a pure VPL, everything can be done within the visual environment and even rely on its graphical representation during the development process.

This does not mean, that a VPL can not be implemented with an already existing programming language, but this system has to work on its own and the user has to be able to create desired applications without leaving the visual environment. If this is not the case and the visual system has merely a supportive functionality, we are talking about *hybrid systems* (see section 2.1.1).

Hybrid systems

Hybrid systems are textual languages with an additional layer of visualization on top. Erwig et al. [8] stated, that visual and textual languages both have their benefits and started to develop *heterogeneous visual programming languages (HVPLs)* by integrating *domain-specific* visual systems into existing *general-purpose* programming languages. They saw visual programming especially superior in working with data structures and focused their research towards this area.

An example they outlined is a visual environment to work with *self-balancing binary search trees* which is fully integrated into the programming language *Pascal* i.e., every interaction with the visual environment gets parsed underneath and the final outcome is pure Pascal code.

Visually supported systems

This category has the least in common with the definition of real VPLs, representing simple graphical aids or clues and mostly integrated into IDEs rather than the language itself.

An example can be found within Microsoft's IDE *Visual Studio* for C# and Visual Basic called *Class View* which visualizes the project's class hierarchy as a graphical tree structure. This is even less than *hybrid systems*, because they do not exist to manipulate or extend the textual language

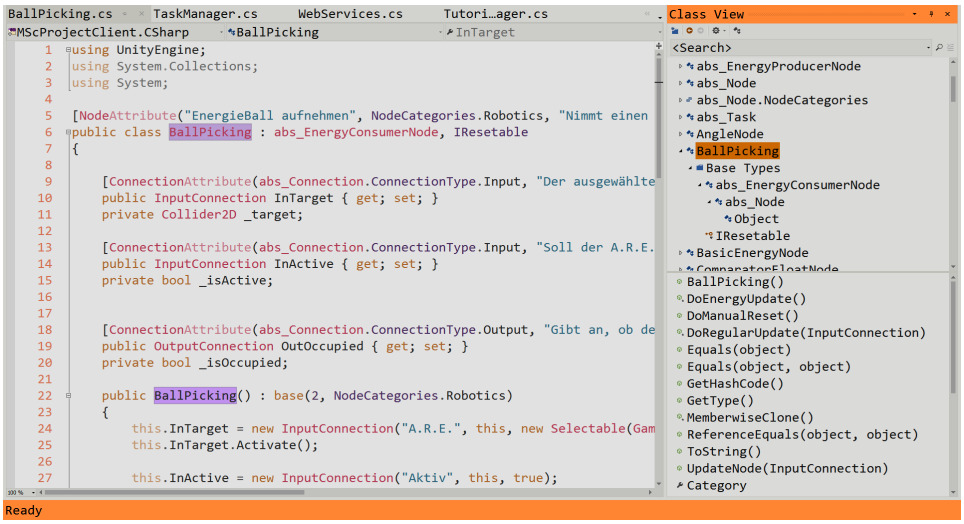


Figure 2.1: A screenshot from the prototype's source code in *Visual Studio 2015*. On the right side you can see Visual Studio's *Class View*, an example for *Visually supported systems*.

through a graphical environment but rather visualize structures which are easier to understand in a figurative way (see figure 2.1).

2.1.2 Common VPL Systems

As there are lots of different VPLs existing today with different purposes and scopes, there are also different ways of how these VPLs are graphically displayed. Brown [3] differentiates between the *semantic* and the *syntactic* base of an VPL, where the semantic base defines, how the system works and the syntactic base, how it is visualized. While one could theoretically try to randomly mix up two bases, there are indeed naturally fitting combinations and even constraints on which combinations do work together.

This section does not explain *semantic* and *syntactic* categories on their own but rather presents a selection of combinations more commonly found in existing VPLs.

Form-based

The idea behind form-based VPLs is to imitate typical spreadsheet applications as they are a popular tool for non-programmers. Although there is no remarkable current VPL based on this visual system, *Forms/3* was fairly popular in the 90s, especially for scientific research and prototyping. *Forms/3* is a *general-purpose, declarative VPL* and incorporates typical aspects of spreadsheets like tables with a matrix of cells which in turn can

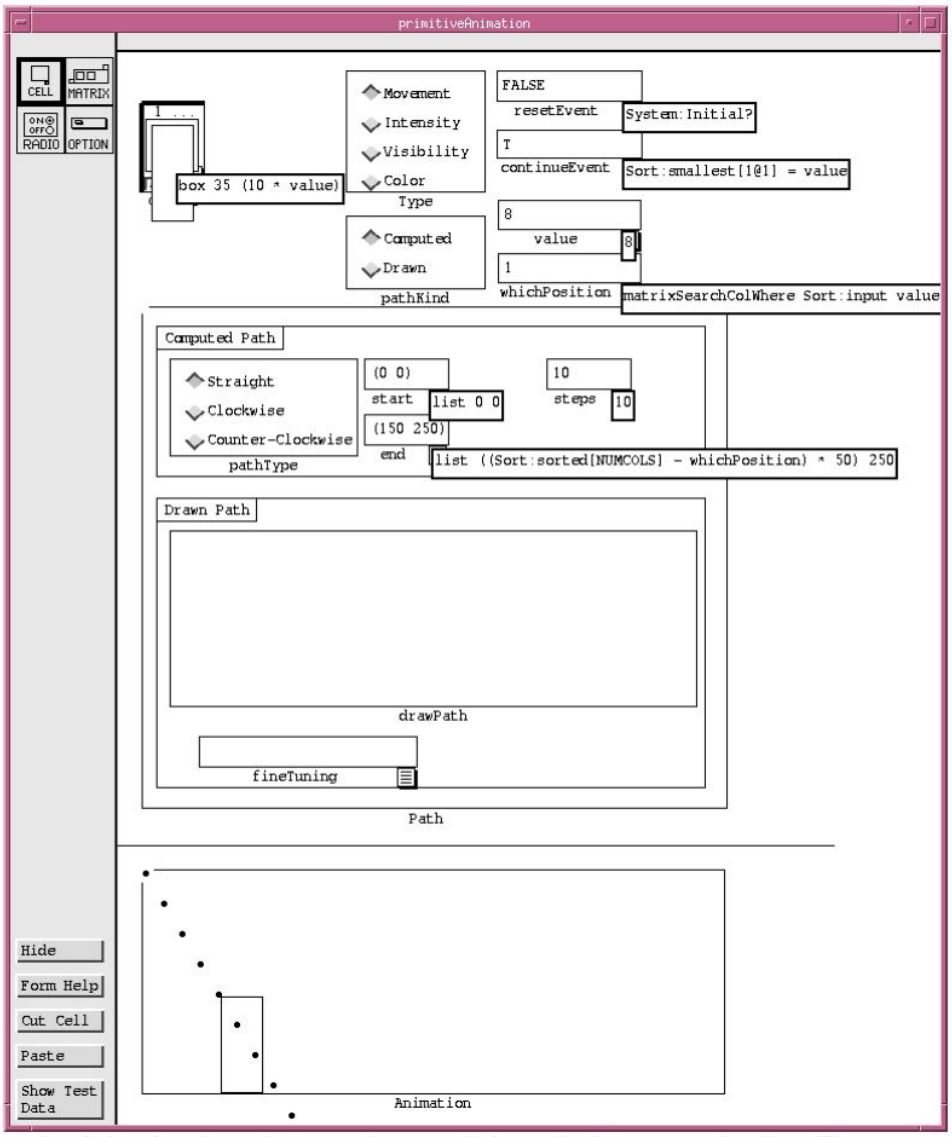


Figure 2.2: This figure shows an example for a Forms/3 program, resembling the look and feel of a typical spreadsheet environment [4].

hold alphanumeric data, images or complex formulas and functions (see figure 2.2). Tables are objects placed within forms — the equivalent to sheets — which then can be placed in other forms again as subforms. Forms/3 is furthermore capable of data abstraction, GUI/IO and animations as Forms/3 uses a three-dimensional computation model with time as third dimension [4]. Unfortunately, Forms/3 is not actively developed anymore since the mid-2000s, but is still a very remarkable example for form-based VPLs.

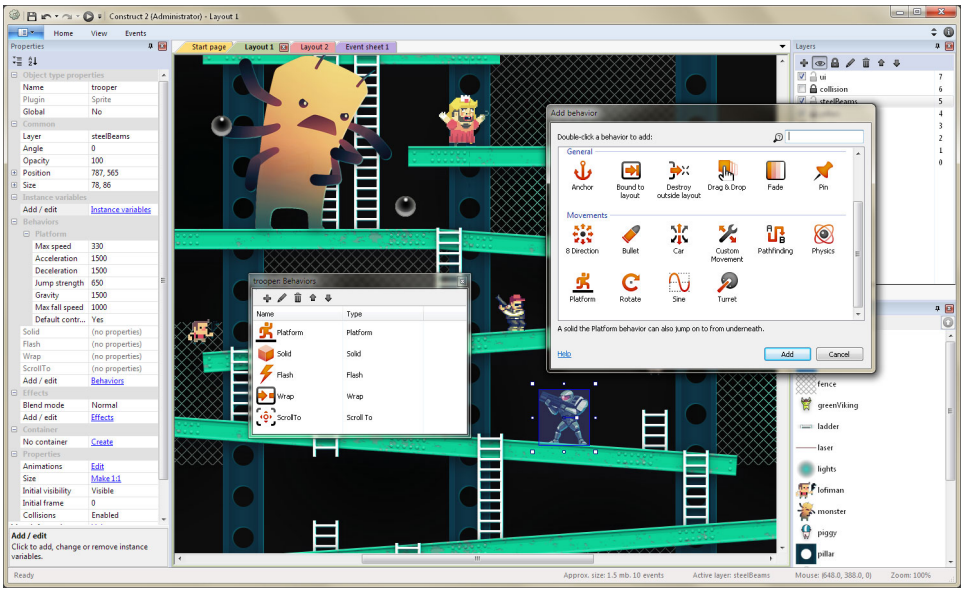


Figure 2.3: Construct 2 with its event-driven component-based drag and drop system [17].

Event-driven Component systems

As the name already indicates, component-based VPLs incorporate the *component-based programming paradigm* where components are defined as independently developed, composable and reusable black-boxes to emphasize the separation of concerns within computer programs [6, 16]. They are gaining a lot of popularity nowadays thanks to a couple of modern game engines like *GameMaker: Studio* or *Construct 2*. Both are based on event-driven systems with large libraries of ready-made components that can be simple added to game objects by drag and drop which in turn extends these objects with additional behaviors (see figure 2.3). Both engines can be extended with additional components through custom *markup languages*.

Sequential Blocks

Programmers are used to read and write software on a line-by-line basis and *Block VPLs* try to imitate this habit to feel more familiar to actual developers and to teach non-developers this way of thinking, thus they are often used for educational applications. One of the earliest examples utilizing this system is *LogoBlocks* [1], a VPL based on *Lego/Logo* which is in turn a MIT research project (and precursor of Lego Mindstorm) to program real Lego robots (see figure 2.4).

Another example is *Blockly*, an open-source library written in JavaScript by Google. It is a web-based visual editor outputting syntactically correct

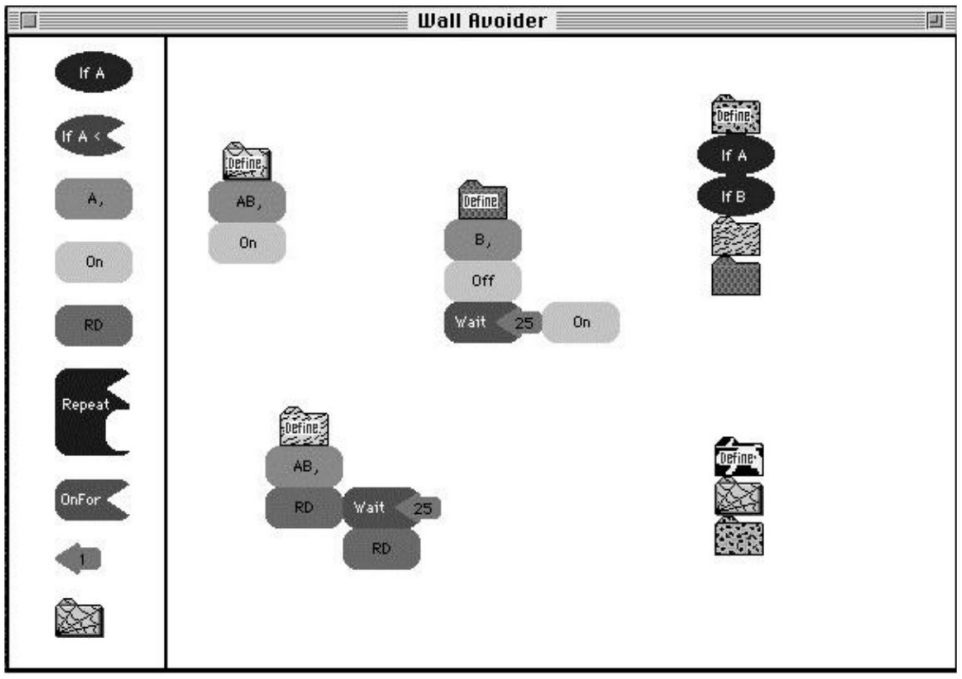


Figure 2.4: An example application created with *LogoBlocks* which enables a real Lego robot to move around without crashing into walls [1].

code, thus it can be classified as a *visual hybrid system* (see section 2.1.1). But since Blockly can be customized and fully integrated into web- and android-applications very easily, there are lots of educational software and games using this framework[1].

A third example is the educational project *Scratch*, where people can create small games and animations within their browser with a set of predefined blocks (see figure 2.5 for an example program). It is interesting to mention, that the developers of Blockly teamed up with Scratch's developers for a side-project called *Scratch Blocks* to re-implement Scratch using Blockly.

Directed Graphs

Directed graphs or *digraphs* are described in the mathematical studies called *graph theory* as a set of *vertices* (or nodes) connected by *directed edges* (or arrows) describing the relation of these vertices. While edges in regular graphs allow both directions, directed edges can be constraint towards one direction.

[1]There is even an acquirable early-access game on the gaming platform *Steam* called *CodeSpells* which uses Blockly to create magical spells. The game can be found on http://store.steampowered.com/app/324190

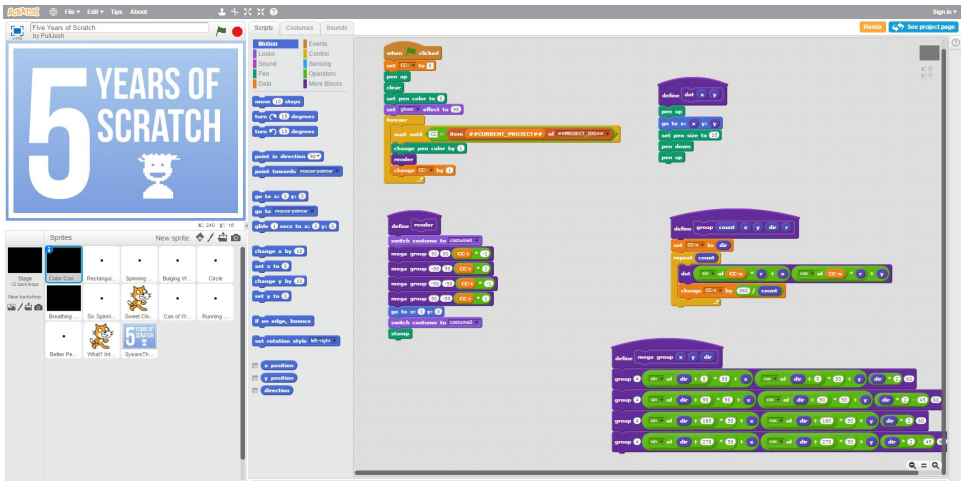


Figure 2.5: This figure shows an animation done with the web-based editor of Scratch. On the left side is the animation and the used sprites and on the right is the editor.

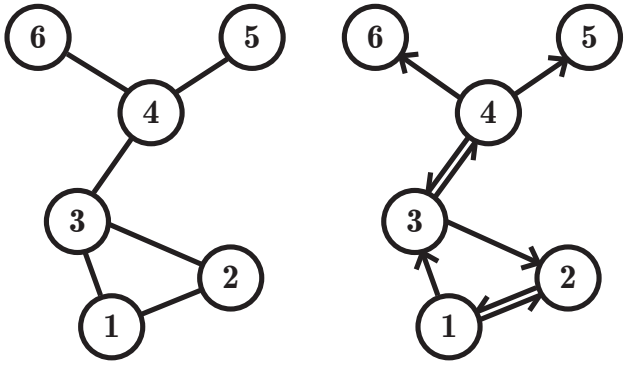


Figure 2.6: An example for a regular graph on the left side and a directed graph on the right side with directed edges showing the direction they are associated with.

Though graphs are indeed a mathematical representation of a set of elements with matrices or lists as notation, they are often described with graphical visualizations, because the relations between the vertices is more recognizable for humans on this way (see figure 2.6).

Digraphs are currently the most popular VPL systems and a natural fitting visual system for the *flow-based programming paradigm* (see section 2.2), therefore it is often called *visual flow-based programming*. It is also used as the basic system for the prototype. Furthermore they are often used for various other real world data set visualizations like financial transactions or transportation routes.

2.1.3 Advantages

As already mentioned in section 2.1, there is a large variety of opinions regarding VPLs, nevertheless they gained lots of popularity over the last years and there is a large range of well-known and very mature projects incorporating VPLs — a few advantages often mentioned are presented now.

Comprehensibility

An influential work regarding visual languages was written by Hirawaka and Ichikawa in 1994 [10], stating:

> Pictures are superior to texts in a sense that they are abstract, instantly comprehensible and universal.

Although many authors argue against such a *superlativist position* of graphics over text, VPLs can indeed be superior regarding readability whenever they help to *explain the inner workings of a system at a glance* [12]. Larkin and Simon [11] came to a similar conclusion when they compared diagrammatic to sentential representations of information, stating that diagrams indexed by location in a plane can help to group information together and thus reduce the effort of searching.

Beginner-friendly

A huge appeal for VPLs is the low threshold to start working with and although everything has some kind of learning curve, VPLs still maintain a fairly low entry and its users are often capable to achieve first successes very early. Furthermore, a few, including [9, 12], also mentioned that this low entry level and the fast results of visual programming environments are also very motivational for beginners which do not have to learn any syntax or concept of regular programming language beforehand.

Beginner-friendly does not only point out an advantage to users new to a certain environment but also to users regarding their general skill level of textual programming. This is a huge advantage and a main reason for its success nowadays especially for *high-level domain-specific environments*. Hirawaka and Ichikawa concluded their work with a similar statement [10]:

> When we use visual expressions as a means of communication, there is no need to learn computer-specific concepts beforehand, resulting in a friendly computing environment which enables immediate access to computers even for computer non-specialists who pursue application.

This is the main reason for a lot of projects to implement visual environments for very specific purposes in a textual programming language by regular developers. These tools are then incorporated into the work-flow of co-workers

without programming knowledge. This is a very common way to streamline the working process of modern companies.

2.1.4 Disadvantages

While there is increasingly more work done in the field of VPL and a multitude of scientific research shows lots of advantages (see section 2.1.3), there are also very critical voices towards VPL, showing various disadvantages.

Low-Level Inefficiency

VPLs providing lots of components incorporating complex functionality indeed helps reducing the amount of work to reach a certain goal. But easy tasks like simple calculations can take some time with visual programming to set up all graphical elements, while this is often not more than a short line for textual programming.

Boshernitsan et al. [2] distinguish between two levels of procedural abstraction, *high-level* and *low-level visual programming languages*, which we can find in regular programming languages too. They mention, that general-purpose VPLs are often found on the low-level side with features like conditionals, iterations and atomic operations like additions and conclude. While this enables these languages to be used within a larger scope, using text is a more appropriate way of such low-level work. Furthermore they notice that lots of low-level operations in VPLs lead to a cluttered display pretty fast, which is also called *Deutsch Limit* (see section 2.1.4).

Deutsch Limit

L. Peter Deutsch, a Boston-born software developer and composer made following assumption during the presentation of a newly developed visual programming language by Scott Kim and Warren Robinett [1, 18]:

> Well, this is all fine and well, but the problem with visual programming languages is that you cannot have more than 50 visual primitives on the screen at the same time. How are you going to write an operating system?

While this limit is not scientifically proven, there is still an advantage of textual programming over visual programming regarding information density — which Deutsch tried to make clear with his statement. Screen space can get a limiting factor soon, especially when lots of visual primitives have to be used for relatively less functionality, as described in section 2.1.4. There is also a large study by Whitley and Blackwell [14] with three different surveys they conducted and then compared. One of them is about *LabVIEW*, a

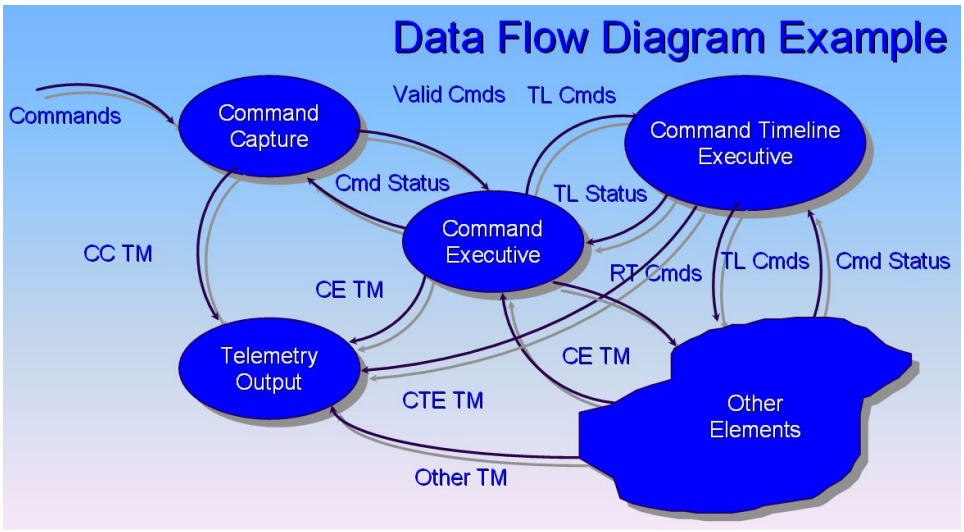


Figure 2.7: This is an example for a Data Flow Diagram taken from a lecture of *John Azzolini* [15].

visual programming environment for industrial automation and system design. The overall conclusion is fairly positive regarding VPLs but especially the survey about LabVIEW showed that users often produced messy and cluttered programs.

However, an argument against the Deutsch Limit is, that this limit could also exist for the textual visualization [1]. Furthermore, in modern visual programming language, this is often no problem anymore, because most of them already have a way to hide blocks of visual primitives in sub-structures, therefore the Deutsch Limit is never reached.

2.2 Flow-Based Programming

Flow-based programming is the core idea behind this thesis's prototype, therefore a more in-depth explanation will be given in this section starting with a short overview of its history.

2.2.1 History

Flow-based programming (FBP) was invented in the early 70's by John Paul Morrison, while he was working at IBM as an engineer. They where working with lots of *Data Flow Diagrams* (DFD), which was a popular way to plan and visualize computer processes these days (see figure 2.7 for an example). But Morrison really struggled to effectively work with them, because DFD where based on very old *Flowchart Planning Processes* used by IBM

since the late 19th century. IBM was called *Computing-Tabulating-Recording Company* and was one of a few companies building *Electric Accounting Machines* (EAM), where lots of physical punch cards where processed through various EAMs — it was the beginning of large-scale data processing. The problem was that they where still working and planning large software applications with DFDs at IBM but had to translate these architectures into ones that can be run on a computer based on the *von Neumann paradigm*.

Morrison really loved the idea of encapsulated blocks with a certain functionality, running at the same time, processing incoming data and proceeding them to the next block. With further influential work like Geoffrey Gordon's *General Purpose Simulation System* [20] or Conway's *Coroutines* [7] in his mind, he finally created FBP. It was used for a few projects at IBM including a software for a Canadian bank, where parts of it are still running. Unfortunately, it never took off, on the one hand because Morrison never really actively promoted this new paradigm, on the other because most of the other programmers did not really want to adopt to this new way of thinking.

Now, forty years later, people start to get interested in FBP again. There are some pure FBP implementations, for example in *C++*, *Java* and *C#*, which can be found on Morrison's website [25]. But most of the best known projects are just FBP-like implementations, as Morrison himself calls them, using only parts of the concepts of FBP and focusing more on the visual node-based approach, like this prototype does.

2.2.2 Definition

FBP is defined as a *Declarative Dataflow Programming Paradigm*, which can again be broken down into 3 well-defined terms:

Programming Paradigm

A fitting definition of the *Programming Paradigm* can be found on Wikipedia [27]:

> The notion of programming paradigms is a way to classify programming languages according to the style of computer programming. Features of various programming languages determine which paradigms they belong to; as a result, some languages fall into only one paradigm, while others fall into multiple paradigms.

In other words, every programming language that exists is classified by the features it provides. Nowadays, most of the programming languages have a set of paradigms they incorporate and often provide native or third party libraries to provide even more of them.

Declarative Paradigm

Programming languages are *declarative*, whenever their users do not need to write *how* something is done. The programmer defines *what* has to be done but does not care how the programming language achieves that. Furthermore, the *how* of the implementation does not manipulate anything outside of its scope, which means that declarative programming is *referential transparent*[2].

A very good example for this way of thinking is found in the *object-oriented* language C# — which is actually the opposite of the declarative paradigm. C# has its own declarative API[3] though, namely *LINQ*. Iterating over a list and picking certain elements by hand is the object-oriented way, while LINQ exposes methods like *Select*, *Join* or *Concat*, which can be simply implemented to do the same, while the user does not have to care about how it is done by LINQ.

Dataflow Paradigm

The focus of *Dataflow Programming* lies on the data of an application. The data is constantly passed around from one stateless instruction to another, like water flowing through pipes. The instructions are seen as "black boxes", which can be interconnected and exchanged to form the functionality of an application. As you may have noticed, we just talked about "black boxes" and "stateless", which indicates, that every programming language incorporating the *Dataflow Paradigm* also incorporates the *Declarative Paradigm*.

A well known language using this paradigm is the *Unix Shell* with its often used pipes. Processes can be chained together by writing a vertical bar between two of them, which means that the incoming data is handled by the first process and the output is automatically processed by the second one.

The following example is a valid code snippet and perfectly shows the idea of the dataflow principle:

```
ls -l | grep something | less
```

The first process *ls -l* gets all files in the current directory, *grep something* goes through this given list and filters all files with a name containing the given string "something" and *less* finally prints this filtered list to the console

While *flow-based programming* is very similar to *dataflow programming*, there are also multiple differences, clarified by Morrison [13, 23]:

> Flow-based programming is a particular form of dataflow programming based on bounded buffers, information packets with

[2] An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program.

[3] API means "Application Programming Interface", a popular way of utilizing third party libraries by implementing clearly defined and well documented functions, that are exposed to the public by the developers.

defined lifetimes, named ports, and separate definition of connections.

2.2.3 Terminology

Morrison really cares about the right use of the terms within FBP. A brief overview of the most important ones is given in this section, a lot more can be found on the official Wiki, which is maintained on *Github* [19].

Graph

Graphs represent whole applications, that define their functionality through networks of interconnected components. To reduce the complexity of large projects, parts of the network can be combined to *Subgraphs* to make the main graph more clean and to collapse certain blocks of functionality into one to probably reuse it again elsewhere. Morrison also distinguishes between a passive graph and an active one that is currently running, which he calls a *Net*.

Component

Components are the heart of graphs, blocks of encapsulated functionality that have no information about the rest of the graph. They are passive and do nothing until they get activated by receiving *Information Packets* on one of their *Input Ports*. As soon as they are running and processing incoming data, they are called *Processes*.

Port

All components have a certain number of named *Input-* and *Output Ports*. A component receives new data on input ports and as soon as all input ports have some data, the component starts to process them. When its done, it proceeds the data to the corresponding output ports and does not care anymore about it.

There are some further special types, which can be found here [19]. One important type should be mentioned though, namely *External Ports*, which are used for embedded subgraphs. With external ports, the developer can expose inputs and outputs for the whole subgraph, which then can be connected to the rest of the graph.

Connection

Processed data that got moved to an output port by the component now has to move on to the next component. To define, which components are connected to each other, *Connections* are made between an output port

from one component to a different component's input port. Connections are *Bounded Buffer*, as already mentioned in section 2.2.2 — in other words, as soon as a connection is full, the process that sends the data blocks and whenever the connection is empty, the process that receives the data blocks.

Information Packet (IP)

Information Packets carry the data that gets moved around between components. They activate a component, when they arrive at an input port, except a special type of IPs, *Initial IPs*, that simply carry some configuration information with them without triggering components.

2.3 State of the Art

Morrison often mentions, that there is a difference between the way *he* defines FBP, which he calls *classic FBP* and most of the modern implementations which are *FBP-like* according to him (see section 2.2.1).

In the following sections, some projects incorporating more or less ideas of FBP are presented, but they are all FBP-like implementations, according to Morrison. Most of them also use different terms, but they all mean the same most of the time. A good way of categorizing them is by their purpose. Like regular programming languages, there are *General-purpose languages* that are suitable to produce programs for different purposes and of suitable size and *Domain-specific Languages* that accommodate particular domains and can not be used outside their scope.

One thing they all have in common is a visual interface though and therefore belong to the group of visual programming languages (see section 2.1). Thus the following examples are *Visual Flow-based Programming Languages* which where also used as references for the thesis's project regarding functionality, interface design and user interaction.

2.3.1 NoFlo

NoFlo is a free general-purpose programming language which was developed in 2012 to fully incorporate the concepts of FBP in *Javascript* without compromises — Morrison [24] still calls it FBP-like and even wrote an article about the reasons that NoFlo probably will never be a true *classic* FBP implementation.

NoFlo itself is no visual editor — it was implemented in *Node.js* and *CoffeeScript* and is a library to develop JavaScript web front- and back-end applications[4]. The terminology is the same as Morrison's, because the developers where heavily inspired by his books. They created an own domain-

[4]You can find the whole MIT-licensed source-code online on Github: https://github.com/noflo/noflo

Figure 2.8: The current NoFlo logo.

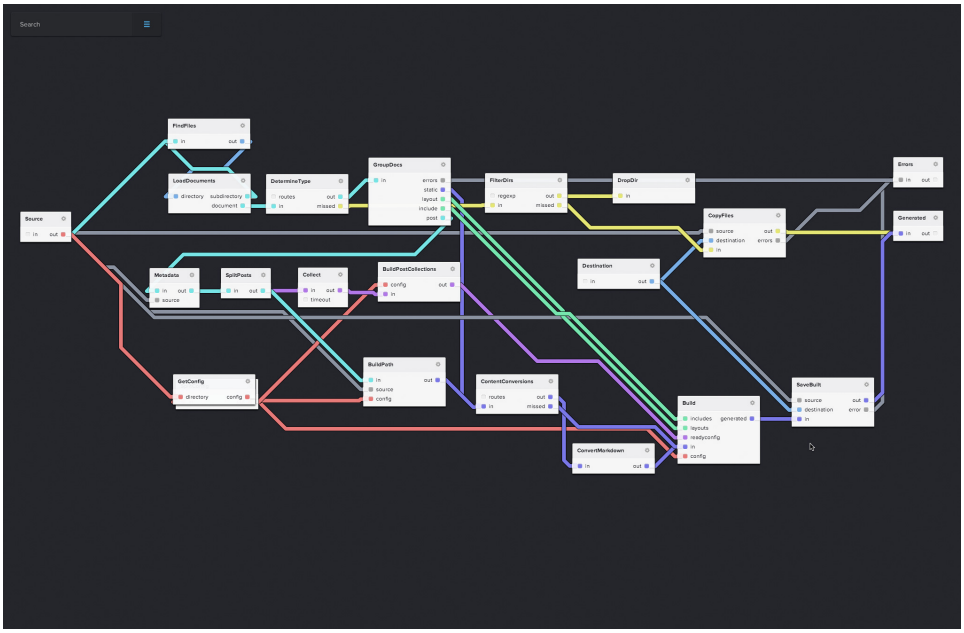


Figure 2.9: Full NoFlo version of the well-known library Jekyll [26].

specific language similar to *JSON*[5] to define the application's graph outside of the JavaScript files. Soon after they released the first version, it became very popular and a large community started to develop lots of ready-made components integrating a large set of different APIs commonly used for web-development — everything managed and distributed over *NPM*, a package manager for Node.js.

When the developers realized the popularity of NoFlo, they started a *Kickstarter* campaign and successfully funded the development of a visual web-based editor for NoFlo, called *Flowhub*[6]. Mobile touch-apps for *iOS* and *Android* are also currently developed. Flowhub as full access to all com-

[5] JSON as described by Wikipedia [22]: "*JavaScript Object Notation* is an open-standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs."

[6] Flowhub can be found on www.flowhub.io

Figure 2.10: Unreal Engine 4 logo.

ponents developed by the community and own components can be easily added with JavaScript. Complex graphs including sub-graphs can be made, analyzed and saved in all current web-browsers. One popular example for the effectiveness of NoFlo and Flowhub is a fully working implementation of *Jekyll*, a "simple, blog-aware and static generator for small websites" with just a few custom components (see figure 2.9).

2.3.2 Unreal Engine 4

Unreal Engine 4 (UE4) is a modern 3D game-engine for AAA games, indie projects, real-time motion-capture pre-visualization in movie production, real-time high-quality visualizations of cars or architecture and is one of the front-runner in bringing *Virtual Reality* to the developer-masses. A wide variety of successful games like *Tekken 7*, *Kindom Hearts III*, *ARK: Survival Evolved* or *Street Fighter 5* are made with UE4.

UE is written in C++ and you can use solely C++ to develop new functionality but since it was rebuilt from ground up in version 4, they implemented a new core system called *Blueprints Visual Scripting System* which is a visual editor for nearly everything in the engine like general gameplay scripting, complex AI behavior and even UI functionality. The terminology is similar to classic FBP, but components are split up in *Nodes*, *Events* and *Vars* and connections are called *Wires*.

The general game development work-flow with blueprints starts with the implementation of new nodes with functionality not yet present and exposed variables which can then be tweaked. New nodes can only be developed

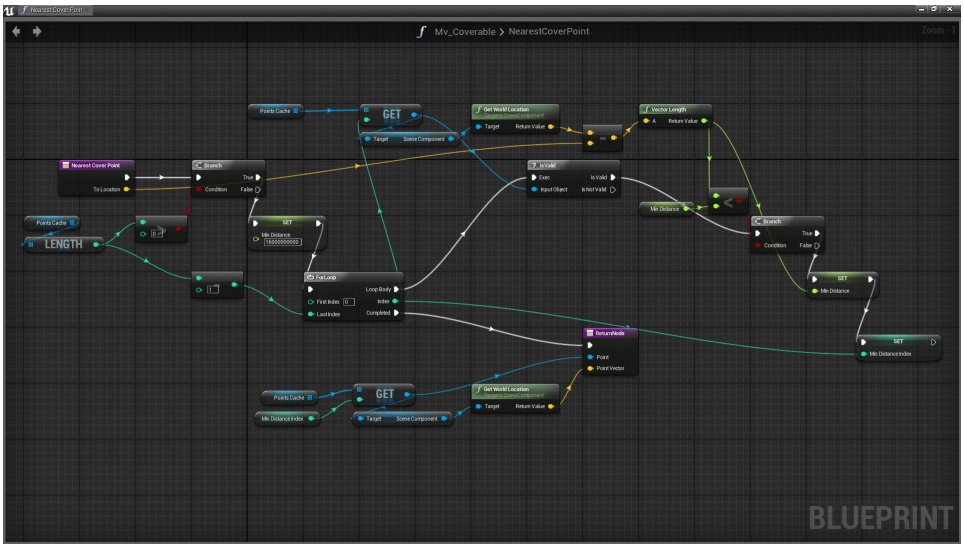


Figure 2.11: An AI behavior blueprint, which enables an entity to find out its nearest cover point in a shooter game [21].

in C++. They can then be used be the rest of the team, especially non-developers, to create new blueprints. Blueprints can be reused and also, like sub-graphs, used within other blueprints. They are then added to game entities to apply their functionality to them. Figure 2.11 shows a blueprint for some AI behavior. With this blueprint, the entity it is applied to, tries to find the nearest cover point in a shooter game.

2.3.3 vvvv

vvvv describes itself as a "general-purpose toolkit" and a "hybrid visual / textual live-programming environment for easy prototyping and development". It is *free for non-commercial use* and written in *Delphi*. It is indeed a general-purpose framework, but the focus is clearly on real-time multimedia manipulation and it is used for *2D/3D Animation*, *Data Visualization*, *Multi-screen Systems* or *Projection Mapping*.

Regarding the terminology, graphs are called *Patches*, components are *Nodes*, ports are called *Pins* and connections *Links*. It has a feature called *runtime only mode*, because everything is compiled on the fly the whole time. New nodes are written in C# with the whole *.NET-API* accessible and even shader programming can be done within vvvv by implementing shader-nodes written in HLSL. Another feature is called *boygrouping* which is a performant built-in client-server architecture which allows to control one server-application to control multiple clients over the network for large-scale multi-screen setups.

Figure 2.12: The current vvvv logo.

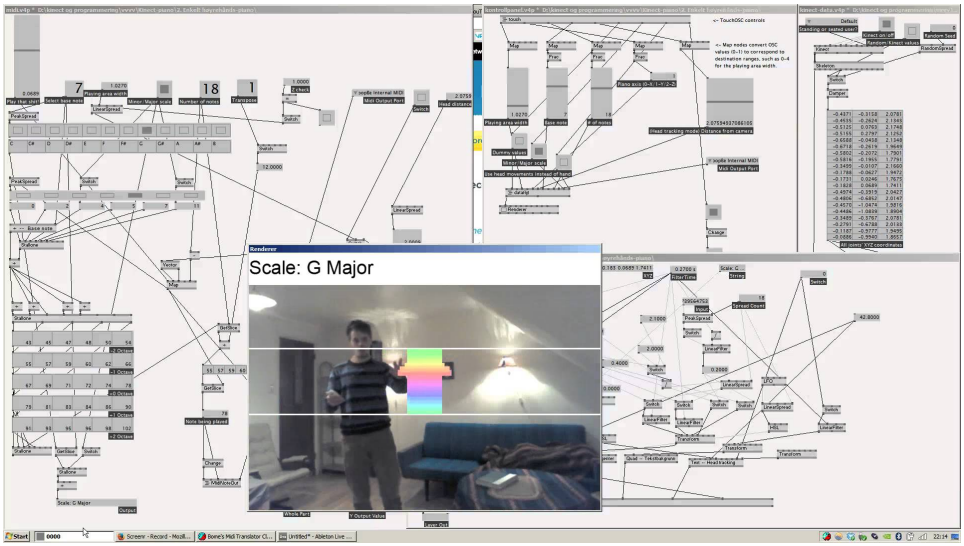


Figure 2.13: An interactive kinect-controlled midi piano made with vvvv by Kristian Peterson. You can see multiple patches and sub-patches and an output window with a live video-feed.

The main reason for vvvv's success is once again their very active community, because nodes and even whole patches can be easily shared on their website which enables vvvv to utilize a huge community-driven library of ready-made nodes and patches. Thus, nodes for nearly every microcontroller or physical gadget exists which made vvvv very popular for developers in the field of *Physical Computing* and interactive (artistic) installations. An example is a project done by Kristian Peterson for the *Oslo Maker Fair 2014*. He created an interactive video-wall allowing people to play a digital midi piano controlled by a *Microsoft Kinect* (see figure 2.13).

Other often used community-made nodes include wrapper for modern 2D / 3D physics engine which enables vvvv to visualize complex physical simulations and wrapper for lots of different database systems.

Chapter 3

Prototype — NO:LE:AP

After a brief overview was given about the core topics of this thesis, this chapter describes the concepts that made it into the prototype, the current status of the prototype and a brief description of things that where added to the prototype after a few tests made with a small group.

The current title is *NO:LE:AP* (**No**de-based **Le**arning **Ap**p) and from now on we are referencing to it as *prototype* or *NO:LE:AP*.

There will be a few figures in this chapter, screenshots taken from the current prototype, where the text is written in German. The decision to release the game in German was made early on while the concepts for the prototype's evaluation where discussed. As there would be no international test users, German was chosen as the prototype's language to remove language as a decisive factor regarding the user tests, which are described in section 3.2.6.

3.1 Early Concepts

The main goal of this thesis is to create a pure visual programming environment (as defined in section 2.1.1) for a domain-specific purpose and due to the author's interest and background in computer games this domain should be settled within given scope — but nothing further was defined in the early stages. The game had to be very stripped down to concentrate on the core elements as there where no other team-members for graphics, sound design or animations.

A fitting setting was found soon after — robotic entities without animations from a 2D top-down view to minimize the graphical effort, furthermore robots are thematically a good choice to be programmable through our node-based editor. An obvious game design choice for robots was a fighting game with robots battling each other in arenas by running through routines de-

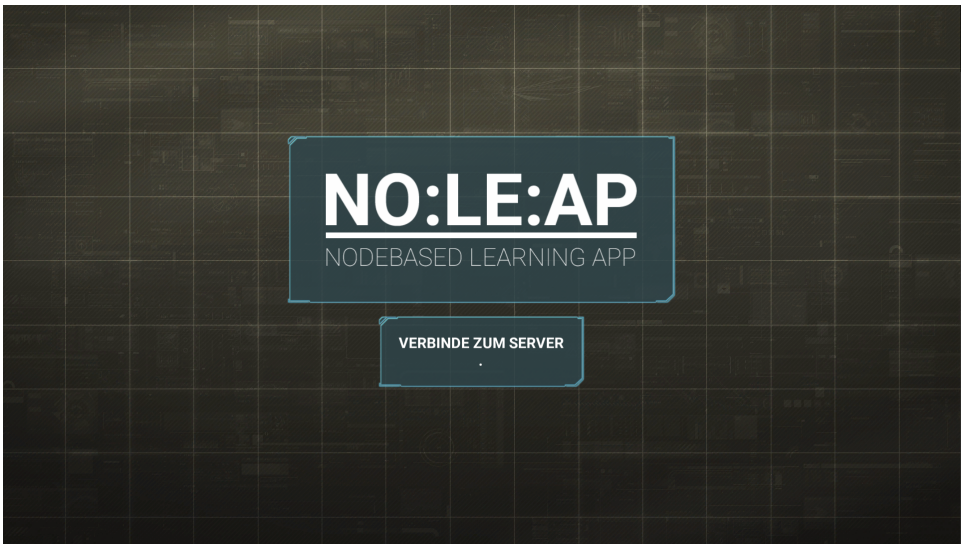


Figure 3.1: Screenshot taken from NO:LE:AP v1.0 while loading and connecting to the server.

fined by a behavior editor. This idea was also influenced by *Robocode*[1], a programming game by Mathew Nelson, where tanks are programmed with *Java* to destroy the tanks of other programmers.

After a few non-relevant tries to further extend this idea, first studies of the state of the art of VPLs showed the still ongoing discussions regarding its efficiency, thus another goal emerged i.e., trying to find arguments for and against it. These arguments also have to be valid and verifiable which implies the need for some kind of performance measurement. Robots battling in an arena did not seem to give enough room to achieve this goal and the author was also not really happy with creating a fighting game. But after a few further iterations, a more suitable idea was found soon.

3.2 Current Version

A more serious approach was taken for the final idea by looking into to scientific field of educational video games. Removing the concept of battle arenas and adding an educational layer brought much more depth to the game and significantly helped to find a suitable way to measure the performance of our node-based editor through player metrics (see section 3.2.6). While the player has still to use the visual editor to program his robot with a certain behavior, his goal is not to kill other robots anymore but to *solve puzzles*.

[1] You can find the latest version of Robocode, an in-depth tutorial and lots of other useful links on the official website http://robocode.sourceforge.net/.

Figure 3.2: This screenshot shows the *world selection screen* with easy worlds starting top left getting progressively harder.

There are now a couple of *worlds*, each with certain unique *tasks* the player has to solve (see section 3.2.4). To make them truly unique, there is a variety of *game entities*, further described in section 3.2.3, that can be utilized for these tasks in various ways. The title for this prototype was found during this phase as well and there was also an early version for a logo created, which you can see in figure 3.1.

As there are now worlds with tasks to solve, the goal was to make each world harder to solve than the previous one, which is a very common but not easily achievable game design choice. To support this progress, the initial idea was to lock all but the very first level and every time the player finishes a world, the next one is unlocked. Unfortunately, this leads to an impassable wall, which immediately prevents the player to see further content, as soon as he does not manage to finish a world. A better way that came up was the following system:

- Whenever a player finishes a world, he gets a certain amount of points credited.
- While new worlds still have to be unlocked to be playable, players can unlock worlds directly by spending points to unlock them.
- To circumvent the just described impassable wall, worlds cost less points to unlock than one can earn by completing a previous one, thus leading to some spare points which can be spent on further worlds, when the just unlocked is too hard.
- To prevent a player to spend points on too hard worlds too early, which

Figure 3.3: After a world is selected, this screen appears, describing the selected world, the tasks to solve and further tips and information.

would again lead to an impassable wall, all worlds are categorized and sorted by their difficulty, bringing the easier worlds to the player's attention at first (see figure 3.2).

Currently, all worlds are divided in three difficulty levels — *easy*, *medium* and *hard*. While easy worlds are mostly used for introducing the player to the game mechanics and to new entities, medium worlds do already take some time to get solved and hard worlds are the most difficult ones, demanding a fairly deep understanding of the systems of NO:LE:AP from the player.

As the player unlocks a world, the only information he gets about it in the level selection screen is the name, the difficulty and the points, the player can achieve (see figure 3.2). After said world is unlocked and selected by the player, a new screen appears — the *world description screen*. This screen shows all available information about the selected world and you can see an example in figure 3.3. On the top left is the name, followed by a small map showing the world and its entities. Underneath the map there is a short summary of a few statistics. The most important area is the text-field on the right side, containing a short description of the world, the tasks to accomplish, available nodes (explained in section 3.2.2) and some further hints.

3.2.1 Network

After the player read through all available information in the world description screen, he can hit the play button in the bottom right corner to actually

Figure 3.4: An actual world of NO:LE:AP, shown as soon as the player hits the play button in the bottom right corner of the *world description screen*.

start the world. figure 3.4 shows the selected world after the player hit the play button in the world description screen. There are also a few buttons in the top edge, starting from left with a *leave world* button, which aborts the current world. Next is a *buy hidden hint button*, showing one hidden hint but also costs a few points (which is further described in section 3.2.6), an *open network* button and finally a *start/stop network* button.

To actually solve the selected world, the player can control all the robots, but only the robots, in the world called *A.R.E.* — further information about them can be found in section 3.2.3. A.R.E.s are controlled through a behavior editor called *Network* which can be opened by clicking on the *open network* button. A network is corresponding to a *graph* in regard to the definition by Morrison already explained in section 2.2.3. The idea is that the player looks at the world, tries to figure out possible solutions for the tasks and then opens the network to start programming one capable of solving given tasks.

Figure 3.5 shows an empty network. On the left side there is a list, holding all available nodes, sorted by their type (which are further explained in section 3.2.2). Underneath this list there are two buttons, *start/stop network* and *close network*. The big empty area on the right side is the network itself.

Networks do **not** run by default, giving the player time to solve the tasks. The world itself is also paused. As soon as he clicks on *start/stop network*, the network grays out, the player *can not modify it anymore* and everything starts running. If something did not work out as intended, the *start/stop network* button has to be pressed again, stopping the network and resetting

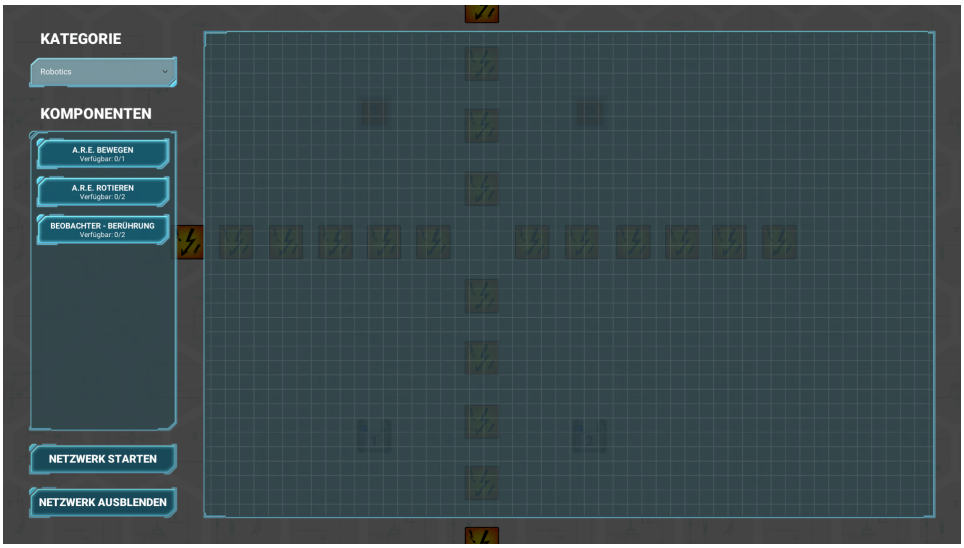


Figure 3.5: Pressing the *open network* button reveals shown screen — the core of the game.

the world to its initial state.

There is also a limited way of navigating within the network available, e.g., pressing the *arrow keys* moves the network around and *scrolling with the mouse-wheel* zooms the network in and out. This especially helps as soon as there are bigger networks to build.

3.2.2 Nodes

Nodes are equal to Morrison's *components* from section 2.2.3. They are passive black boxes and contain bits of functionality. They have input- and output-ports and get activated as soon as some IPs do arrive at their input-ports.

There is a certain amount of certain nodes enabled for the current world to solve the tasks. You can find an entry button in the list on the left for each available node including the number left for this type. These buttons can be moved with drag & drop and dropping one within the network area results in the creation of a node of given type. Following the rules of FBP, output-ports can be connected to input-ports from other nodes — as long as they share the same data type (see section 4.1.2). Figure 3.6 shows a fairly complex network with multiple nodes connected to each other.

Furthermore, there are different colors for different nodes, representing *categories* (regarding their functionality and usage) and all of them are assigned to one. On the one hand, these categories are used to separate the available nodes into different lists and the user can select a category in the

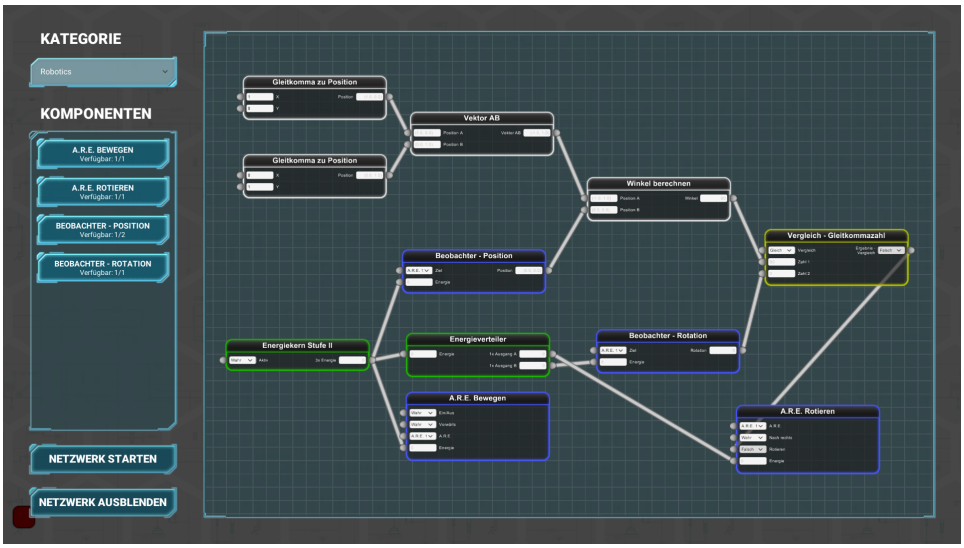


Figure 3.6: Screenshot of NO:LE:AP's network editor with a fairly complex network.

network screen with the drop-down in the top left corner, thus the list is not cluttered with nodes. On the other, different colors for different categories try to support the player working with the network as they give certain visual clues regarding their functionality. Furthermore, not only the nodes are colored but also the connections between two nodes do light up in the color of the output-port's node category, whenever an IP is sent.

As of version *1.0*, which is the version described by this thesis, there are 5 different categories with a total of 24 different nodes — all of them are used for at least one world in the current version. This is a huge variate of nodes which was fairly easy and fast to achieve thanks to the system's implementation (see section 4.1.2). Following, a brief overview of the categories and a selection of nodes will be presented.

Energy

Energy came up as a gamified and stylized alternative to classic update ticks in a game loop, better fitting to the game's setting with robots. Nodes from this category are colored in *green*. The idea behind this system is, that all nodes that interact with the world need energy to work — these nodes can be found in the category *Robotics* and are further described in section 3.2.2.

Energy Core: This is the most important node of all which also comes in three different versions — core level I, core level II and core level III — getting more powerful with increasing level. While regular output-ports are

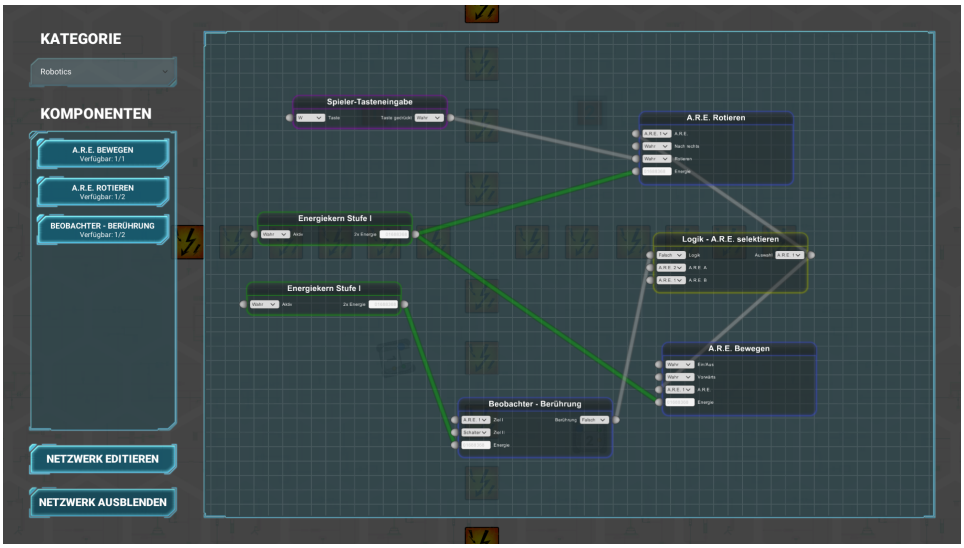


Figure 3.7: A running network with active connections colored in the type of their output-node.

not restricted to a maximum number of connections, energy cores can only supply a certain amount of robotics nodes, which also increases with the core level. They produce a certain amount of energy which gets distributed to the connected nodes, allowing them to do their work.

Figure 3.7 shows a running network with two energy cores level I providing energy to multiple other nodes (which results in the three green colored connections as they reflect the type's color of their output-node).

Energy Splitter: As there is a restriction to the number of nodes that can get connected to an energy core, *Energy Splitter* help to provide energy to even more nodes by equally dividing incoming energy from a core to two further nodes. This results in one more active node but also in less energy for both nodes connected to the splitter. You can see an energy splitter in figure 3.6.

Energy Toggler: This node allows the player to toggle one energy input between two nodes — allowing to enable one node at a certain time while simultaneously disabling the other.

Robotics

Robotics nodes are the blue nodes in figure 3.7 and 3.6 and their purpose is to interact with the game world. There are nodes to interact with A.R.E.s, nodes to survey certain things in the world and multiple further nodes rep-

resenting the player's i.e., the robot's senses. Robotics nodes do all need energy, i.e., all have an input-port for energy and they do only work as long as they are provided sufficient energy. They also do require different amounts of energy to interact with the world — the author's rule of thumb is more energy is needed for interactions that feel more demanding.

A.R.E. Movement: This is an example for the available nodes to directly manipulate A.R.E.s in the game. As the name already clarifies, this node moves a selected A.R.E. around. There are four different input-ports,

1. the default energy input, providing energy to the node to stay active,
2. an input to additionally enable/disable the node (which results in the selected A.R.E. not moving while there is still energy provided),
3. a boolean flag called *forward* which tells the node to move the A.R.E. forwards when true and backwards when false and
4. a list of available and selectable A.R.E.s in the world.

Note that this node does only move the robot forwards and backwards, there is a dedicated node called *A.R.E. Rotation* which enables the robot to rotate.

Position Tracker: While not directly manipulating the world, this and several other nodes have a surveilling purpose. With a *Position Tracker*, a target can be selected (all entities not only A.R.E.s), the actual position of the target is regularly updated and can be retrieved through an output-port.

Energy Ball Picker: With this node, an A.R.E. is capable of picking up energy balls, as long as this node is active and no energy ball got picked up already. This enables the robots to not only move energy balls around but to also place them onto another corresponding entities called *Energy Sockets* which are described in section 3.2.3.

Logic

This category includes all nodes that are used for logical conclusions. While most of the worlds with difficulty *easy* only utilize energy and robotics nodes, difficulty *medium* starts to introduce logical nodes. Their color is yellow, both figures 3.7 and 3.6 show a few of them. These nodes are less high-level and often simply evaluate a given assumption, leading to one single output which states if this assumption is true or not. Still, these are very valuable nodes.

Logical Negate: A very simple node with one boolean input-port and one boolean output-port. Every input gets converted into the opposite.

Float Comparator: This node has three input-ports, two floating point numbers and a comparator which is a list of typical logical operators like *greater than*, *equal* or *less or equal than*. The number from the first input is compared to the second, e.g., $2 < 1$.

Math

As with logical nodes, nodes from the category *math* have increasing relevance with harder worlds. Navigation in a 2D world requires at least points in 2D space and angles, thus there are mainly vector and angle calculations covered by this category. The category's color is defined as white. You can see some math nodes in figure 3.6.

Distance: As the name already indicates, given two 2D points, this nodes results in the distance between them.

Vector AB: Similar to the node before, the name is pretty self-describing. Given two points, the vector pointing from $\boldsymbol{a} = (x_a, y_a)$ to $\boldsymbol{b} = (x_b, y_b)$ is calculated i.e.,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x_b \\ y_b \end{pmatrix} - \begin{pmatrix} x_a \\ y_a \end{pmatrix}. \tag{3.1}$$

Other

The last category, colored in purple, contains all nodes not fitting in any of the previously presented categories. At the moment, there is only one node in this category.

Key Input: While the focus is still on the idea of networks running on their own and that the player is not capable of influencing the network he created as soon as he starts it, there made it one node into the game, that breaks with this rule. While initially developed for testing purposes and to find out how far the implemented system can go, the *key input* node enabled an additional unique layer of interaction for the player and thus was kept.

As the name already implies, this node listens to key input from the player's keyboard. You can see this node in figure 3.7, there is a list of available keys as an input-port resulting in a boolean output-port which is true, whenever the key is pressed. This allows several interactions between the player and the running network, like enabling and disabling robotics nodes.

(a) (b) (c) (d) (e) (f)

Figure 3.8: All entities from the current version of NO:LE:AP. A.R.E. (a), Trigger (b), Jumper (c), Danger (d), Energy Ball (e) and Energy Socket (f).

3.2.3 Game Entities

There are currently six different game entities in NO:LE:AP. Figure 3.8 shows the graphical representations of these entities (also called *sprites* in game development) with a few graphics more matured than others. Entities are used to build the game worlds — nodes as-well as tasks rely on them.

A.R.E.

Figure 3.8(a) is the robot controlled by the user, called *A.R.E* (**A**utonome **R**oboter **E**inheit — which is German for *Autonomous Robotic Unit*). This is the core entity as there is at least one A.R.E. in every world and they are the only way, users can interact with the worlds to solve given tasks.

Trigger

The second entity shown in figure 3.8 is a trigger that can be activated by an A.R.E. through physical contact. As long as an A.R.E. is touching a trigger, the trigger is colored green and active, but whenever the contact is lost, the trigger is disabled immediately, switching its color back to red, showing the user, that it is not active anymore.

Jumper

There is only one purpose for jumpers i.e., to switch their position in the world randomly. They do not move but rather change their position immediately and the next destinations are evaluated beforehand to *only allow positions near the worlds border* (which is a restriction to the randomness that came up after a few tests to eliminate positions that are reachable too fast).

Danger

Danger fields have to be avoided at all costs by all A.R.E.s because as soon as there is a contact between a danger field and an A.R.E., the whole world is reset and the player gets a penalty that is described in section 3.2.6.

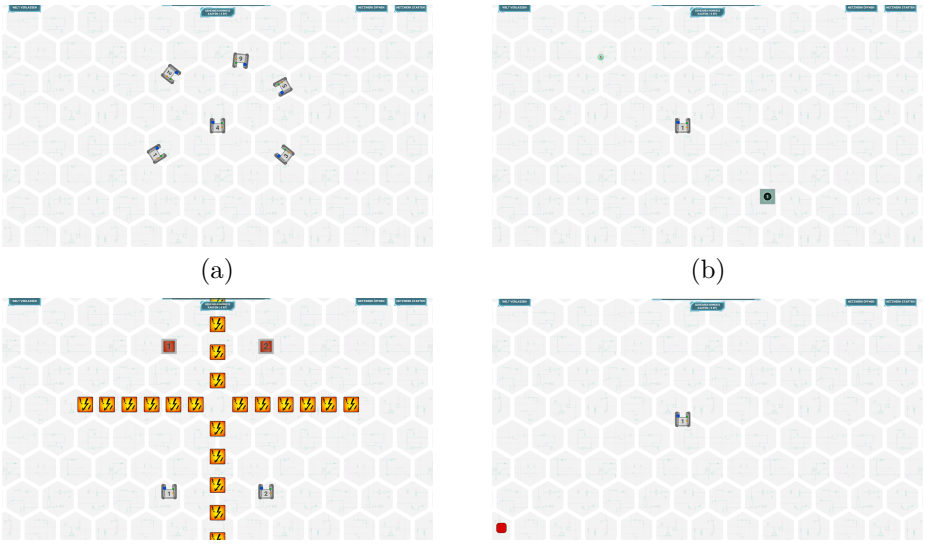


Figure 3.9: Few worlds from the current version of NO:LE:AP. Select and Rotate (a), Energy Ball II (b), Attention, Danger! (c) and Catch the Jumper II (d).

Energy Ball

Energy balls (shown in figure 3.8(e)) only appear with *energy sockets* together and have a unique way of interaction with A.R.E.s as they can be collected and moved around by the robots. A.R.E. can interact with only one energy ball at a time.

Energy Socket

Energy sockets can be activated by contact, similar to triggers. But they are not enabled by A.R.E.s but rather through energy balls placed onto them. As long as an energy ball is placed on a socket, it stays activate.

3.2.4 Worlds and Tasks

At the moment, there are five easy, four medium and one hard world available to play and each of them contains one or more tasks. Currently, there are two main tasks available, rotating an entity for a certain amount of degree and letting two entities touch each other. As soon as all tasks are accomplished *simultaneously*, the player has completed the world. To demonstrate the various task combinations and mechanics in the game, four worlds will be described now.

Select and Rotate: This is the second world to play with difficulty easy and is part of the tutorial worlds, trying to explain the core mechanics to the player. As you can see in figure 3.9(a), there are multiple A.R.E.s placed and the only task is to rotate the robot with the number *four* for 360° degree. The goal of this tutorial world is to present a few new nodes to the player and to show him that he is capable of specifically selecting entities in the world.

Energy Ball II: Not describing any further easy worlds, this is one of the medium worlds to solve. There are three entities in the world, one A.R.E., an energy ball and an energy socket. The only task given is to bring the ball to the socket. Notice the *II* after the name — this is because there is a easy world called *Energy Ball I*, containing the same entities and the same task. While the first world aligns the three entities in one line, thus requiring the player to just pick up the ball while moving the A.R.E. forward, the second world distributes the entities in a more chaotic way. The player has to move to the ball first, then pick it up and finally somehow manage to move to the socket.

Attention, Danger!: Another world with difficulty medium and an example for worlds with multiple tasks i.e., touching the trigger with the number one with A.R.E. one and simultaneously touching trigger two with A.R.E. 2. Notice the danger fields placed in cross-shaped way in the middle of the world, requiring the player to navigate both A.R.E.s around them. The world is shown in figure 3.9(c).

Catch the Jumper II: This is the only hard world currently available, also a second version of an already existing world, as you can see from its name. There is only one task — touching the red jumper with the only A.R.E. in the world. While the first version contains multiple very useful nodes to accomplish this task (which is why it is classified as easy world), this one has only very low-level nodes, forcing the player to do lots of mathematical calculations to actually catch the jumper.

3.2.5 Persistency

Another important aspect that made it into NO:LE:AP already came up during early stages of the concepts, when the question for the type of enemies in the battle arenas arose. There where two ideas discussed, both leading to crucially different games. One the one hand there was a kind of AI system, making the game to a singleplayer experience where the player fights against the computer. On the other hand, the idea of having some kind of persistency came up. Players could create battle behaviors which where then stored on a server and other players could match their robots against robots

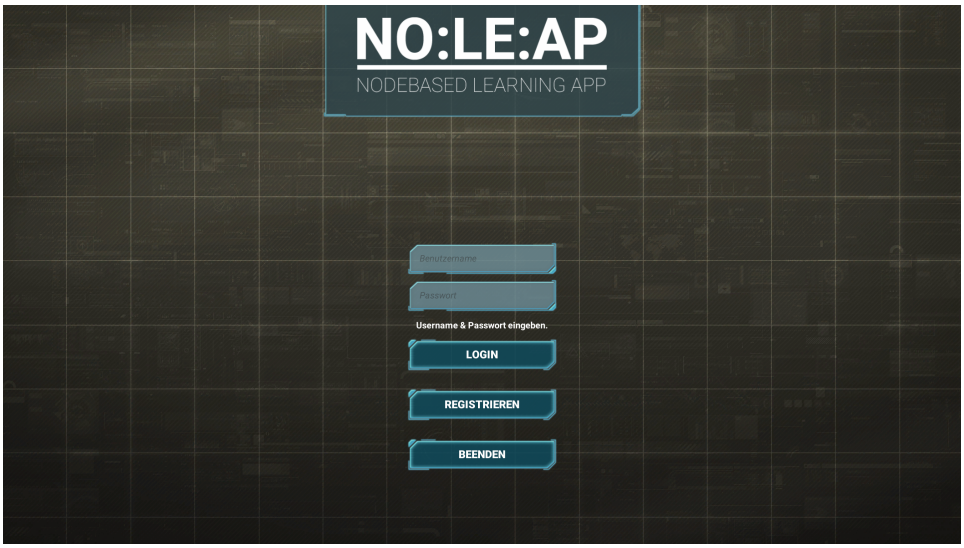


Figure 3.10: The *login screen* from NO:LE:AP.

from other players — similar to the previously described Robocode, where other programmer's robots can be downloaded and battled. While the battle aspect vanished, the idea of persistency is more relevant than ever since there is now a progress regarding earned points and unlocked worlds. As there are already a couple of worlds available, players would loose all their unlocked ones, whenever they close the game. Furthermore, some kind of performance measurement is contemplated, thus the idea came up to store the player's progress online in a database — a technical explanation of the implementation is given in section 4.2. This allows the player to continue to play whenever and wherever he wants to, the only prerequisite is some kind of registration mechanism to store login data in the database for a later recognition.

The current version of NO:LE:AP does have a working login system, a new player has to register himself first and than retrieve his personal progress by typing in his login data. The only minor drawback with this system is the requirement to be connected to the internet to play the game, which should not be a problem nowadays though. There are not only the available points and the unlocked worlds synchronized but also further information like the amount of points left, the player can retrieve from the unlocked worlds. As there are multiple factors of how *well* a world was completed — which is important for and further explained in section 3.2.6 — players will often get penalties on their received points. As with the current version of NO:LE:AP, every world is worth 100 points i.e., whenever the player solves a world in an ideal way, he gets the full 100 points. If he made some mistakes and did not get the full 100 points, he can simply select the just accomplished world and

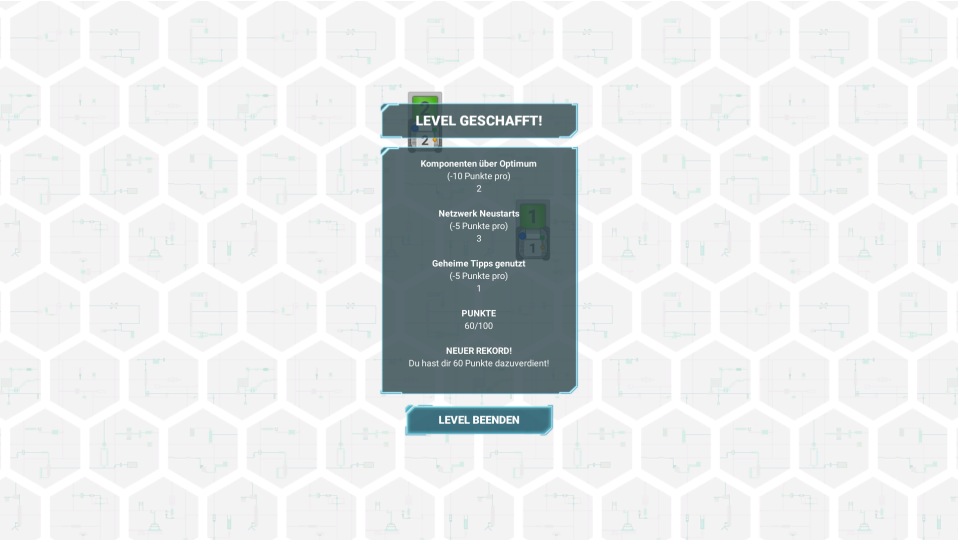


Figure 3.11: The *summary screen* showing the final score of the just accomplished world.

play it again. To prevent a player to gather more than a total of 100 points in one world, the remaining points for every world are stored online and updated after every successful try. So whenever a player starts NO:LE:AP, the first screen he is shown is the login screen which is shown in figure 3.10. After successfully connecting to the server, his progress is retrieved and the game switches to the world selection screen, where the player can continue to unlock and play worlds. Whenever he finishes a world, there is a *summary screen*, calculating the points he accomplished and the actual points he got credited — figure 3.11 is showing this last step before returning to the world selection screen.

3.2.6 Metrics

As there is now a way to recognize the currently playing user, this system can be further developed to automatically gather player metrics in the background while playing. To get valuable data from the user, there are two important categories of information that have to be collected i.e., *who is playing* and *how is he doing*. The first category is called *demographic data*, the second one is the actual *user performance*, both are described in the following sections. This section gives a brief overview of what is collected, an evaluation can be found in chapter 5.

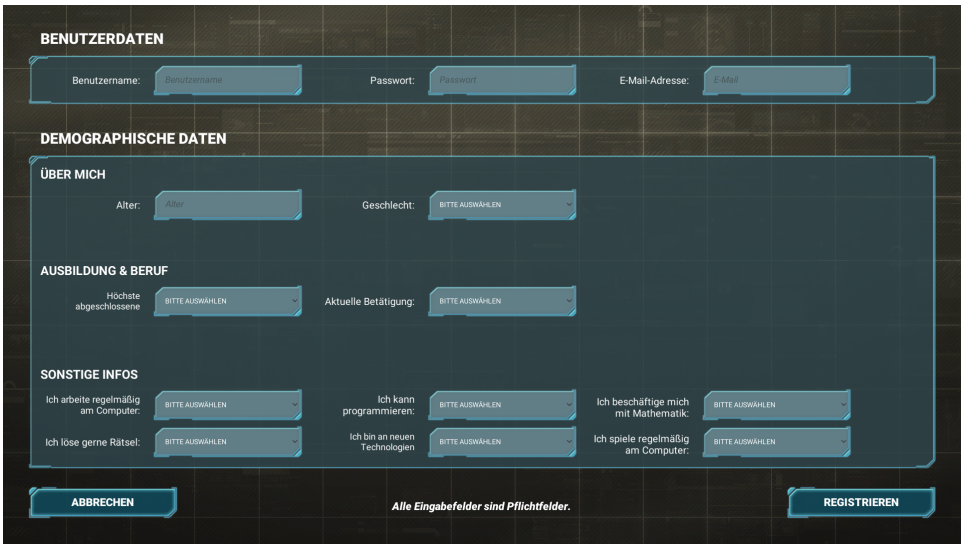


Figure 3.12: The *registration screen* from NO:LE:AP.

Demographic Data

To be able to evaluate the thesis's theory, the user metrics have to be categorized by the player's experience and knowledge with areas like gaming or programming. As the prototype will be spread and people playing it without further observation, this has to be somehow automated. The solution is to use the registration process, which every player has to do and extend it with a few demographic questions about them. Figure 3.12 shows the *registration screen* that appears when a new player clicks the button *register* shown in figure 3.10. The upper panel contains login data like username and password. The lower panel contains the demographic data. Starting with age and gender, the second and third question is about the player's education and current work. The last few questions are about his experience in various areas. Limiting this short survey to a single page should provide enough information to categorize the just registered person while taking not too much time or even stop people to actually complete the registration process. As there are no questions asked about sensible information like name or address, everything collected is fully anonymous.

User Performance

The most important data collected by NO:LE:AP is the actual performance of the playing user. There went a fair amount of work into the definition with a few iterations during the conceptual phase of this project. Following are important metrics collected by the current version of the prototype — more information can be found in chapter 5.

Used Nodes: Every world is tested by the author extensively to find out an optimal solution i.e., the lowest number of nodes needed. While this always depends on the types of nodes provided, there can be multiple solutions, especially starting with medium worlds as there are more nodes available than needed for the optimal solution most of the time. The number of nodes used for the player's network is stored and compared to the ideal solution. The difference is further used for the calculation of the earned points as every node more than the optimum counts as ten points penalty on the final score.

Network Resets: As the player has to deliberately start the network and stop it if it does not work out as intended, this is an important metric to track. It can point out the player's capability of finding a solution without going the way of *trial & error*. Like used nodes, this does also influence the earned points as every restart subtracts five points from the maximum points.

Hidden Hints taken: The *world description screen* (shown in figure 3.3) does already offer some *free* tips before the player starts the world. Another way of helping the player to not get stuck with certain worlds are *Hidden Hints* which he can unlock hint by hint by clicking the button in the top-mid of the ingame screen. These hints are more useful than the free ones and give a good indicator for a player having problems with the current world. They also do come with five points penalty per unlocked hint (which is clearly communicated to the player beforehand).

Taken Time: This metric does not influence the final score but is another very interesting indicator for user performance tracking. There are several reasons for a player taking very long for a world but it is still valid for an overall look at the speed of a player when solving them.

Finished Worlds: All metrics above are collected on a *per-finished-world-basis*, which means that every time a player starts a world and is able to accomplish it, this try gets explicitly stored. As players are able to play a world multiple times, whether to finally solve it for the first time after a few aborted tries or to achieve a better score and to earn the remaining points, it is important to keep track of the different tries and how well he did.

Aborted Worlds: Another interesting information to track, counting every started world that gets aborted either through the *quit world* button or whenever the player closes the application when ingame.

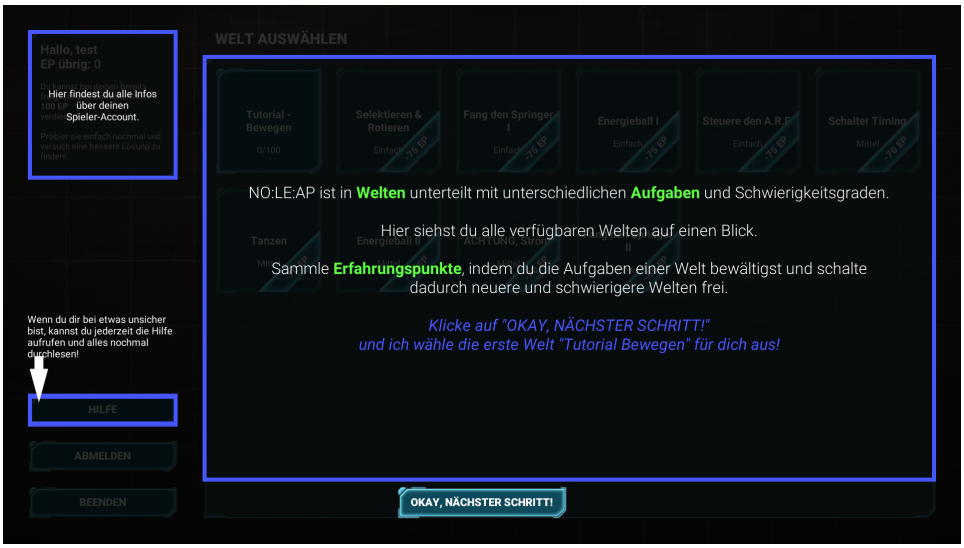


Figure 3.13: Screenshot taken from NO:LE:AP v1.0 showing one of the steps during the tutorial.

3.3 Early Playtest Results

As the game grew in complexity and the tracking of player metrics was already implemented, the idea came up to start an early playtest with a very small closed group which was then not allowed to count towards the valid data collected. The reason for this was to gather first feedback and get rid of bugs while not influencing the outcome of the collected data as early tester already knew the mechanics and most of the worlds. This would lead to distorted data and should be avoided at all cost. The playtest was done with two people and was very successful, a few of the additions that where implemented as a result of these tests are presented now.

Tutorial

The tester where given no clues and no help, they where just observed while they started the game for the very first time. While the author tried to create a structured and logical user interface, it became clear very fast that there has to be some kind of tutorial. This is even more important as the game is planned to be released to a broader group of people who do not get any help.

As the tests showed the necessity for such a system, the current version of NO:LE:AP has a tutorial which starts with the very first login of a new user, explaining every screen step by step and leading the player through the first world. A second round of short tests with the same group resulted

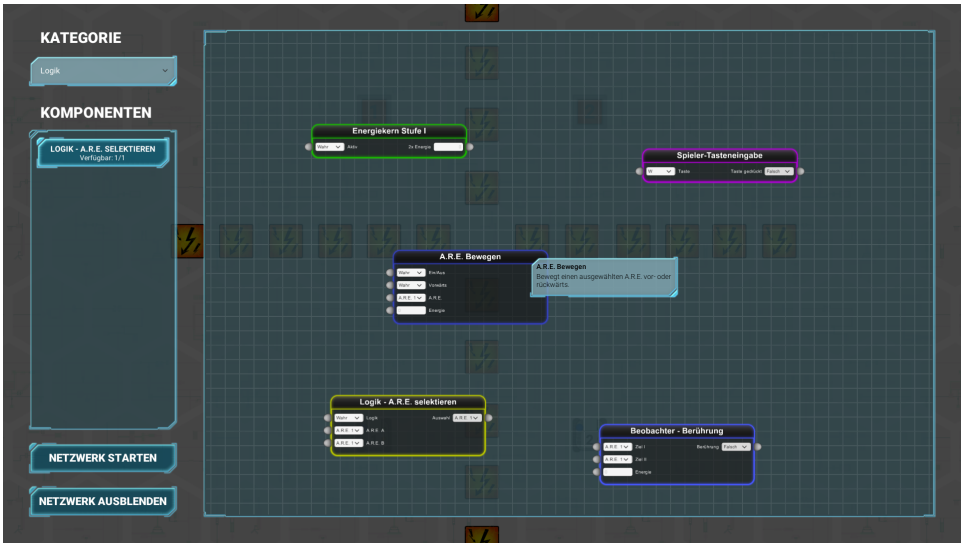


Figure 3.14: As a result from the early playtests, tooltips where added to further explain the functionality of the nodes.

in very positive feedback to this tutorial system. Figure 3.13 shows one of the several steps of this introduction.

Tooltips

As the network editor is the core of the game and the most crucial part to work properly, the testing of this area was especially extensive. While all nodes are categorized and have a well thought out name, it soon showed that this was not enough for the player to fully understand their functionality. This resulted in lots of network restarts as the players had to drag the nodes into the network and start it a few times to see what they actually do.

The solution is to extend not only the node itself but also all input- and output-ports with further descriptions. This additional information is shown as tooltip, whenever the player hovers over the name of the node or the port. Figure 3.14 shows an example for a tooltip and the second round of closed tests showed that player where able to understand the functionality of new nodes much better.

Aborted Worlds

Fortunately, this flaw of the tracking system came up during early playtests, because it would have been capable of ruining the collected data. One of the testers had a few problems with a certain world, had to take a few hidden hints and already restarted the network several times. He already had lots of information about the game and its system due to several talks to the

author. So he knew that the game would sync this poor try as soon as he finishes the world. To circumvent this, he simply quit the world, started it again and was then able to get 100 points on the first try, because he found out the solution before by trial & error. To get rid of this, the already described metric *Aborted Worlds* was added — another way of playing the world without getting a poor try uploaded to the database is to close the whole game, which gets also detected by the current version of the game.

Chapter 4

Implementation

After chapter 3 focused on the concepts and game design decisions forming the actual game, this chapter gives an in-depth view of the technical implementation of the prototype. There are two systems, the client and the server, that work together but run on their own, thus are explained separately.

4.1 Client

The client is the actual game the player downloads and starts — the current version is running on *Windows* only, other platforms are not planned at the moment. To spread NO:LE:AP to as many people as possible, a small web-

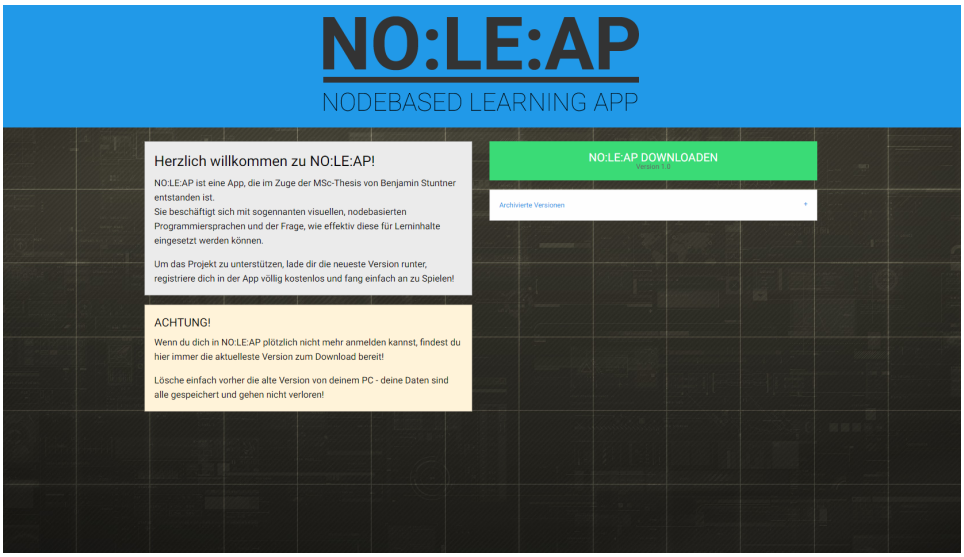


Figure 4.1: The website hosting the download-link of the current version of NO:LE:AP.

site[1] was implemented hosting the download-link of the current version of NO:LE:AP. This is further important because there is a *check for the client's version*, whenever the player is starting the game — a safety mechanism to prevent players to login with an outdated version and probably corrupting the valuable data on the server. As soon as there is an update to the game and the version of the client is different to the version of the server, the game immediately switches to a dedicated *error screen*, telling the player to go to the website and to download the latest version.

This section gives a brief overview of the software used to develop the client and a technical description of the most important parts of the project i.e., the core of the node system, the visual network editor and its connection to the node system and the task system.

4.1.1 Software Stack

Unity3D

The game engine taken for this project is *Unity3D*. The reasons to choose Unity is that there are extensive features to produce not only 3D but also 2D games and apps, including an advanced UI system. Furthermore, it is very fast for creating prototypes and finally the author has already experience in working with it. Unity has a visual editor and a component-based system, with new components written in C#, allowing programmers to utilize most of the .net-framework — with some restrictions depending on the platform the game will be distributed.

Visual Studio 2015

Microsoft's Visual Studio (VS) is one of the most comprehensive IDEs currently available with lots of features like a fully-fledged code completion system called *IntelliSense*, in-depth code metrics, a deeply integrated debugger and the possibility to directly compile and build applications. While VS is not used to build NO:LE:AP, there is a deep integration into Unity with a extension called *Visual Studio Tools for Unity* that enables VS to write C# code for the Unity project.

Adobe Creative Cloud

Adobe with its Creative Cloud is the leading authority regarding software for creative minds. It consists of a variety of specialized apps ranging from Photoshop for image manipulation, After Effects for video post-processing or Illustrator for drawing vector graphics. Especially Photoshop and Illustrator where used to create the graphics for this prototype.

[1] As of date of this thesis is written, the link to the website is http://noleap.tk — figure 4.1 shows a screenshot of the latest version.

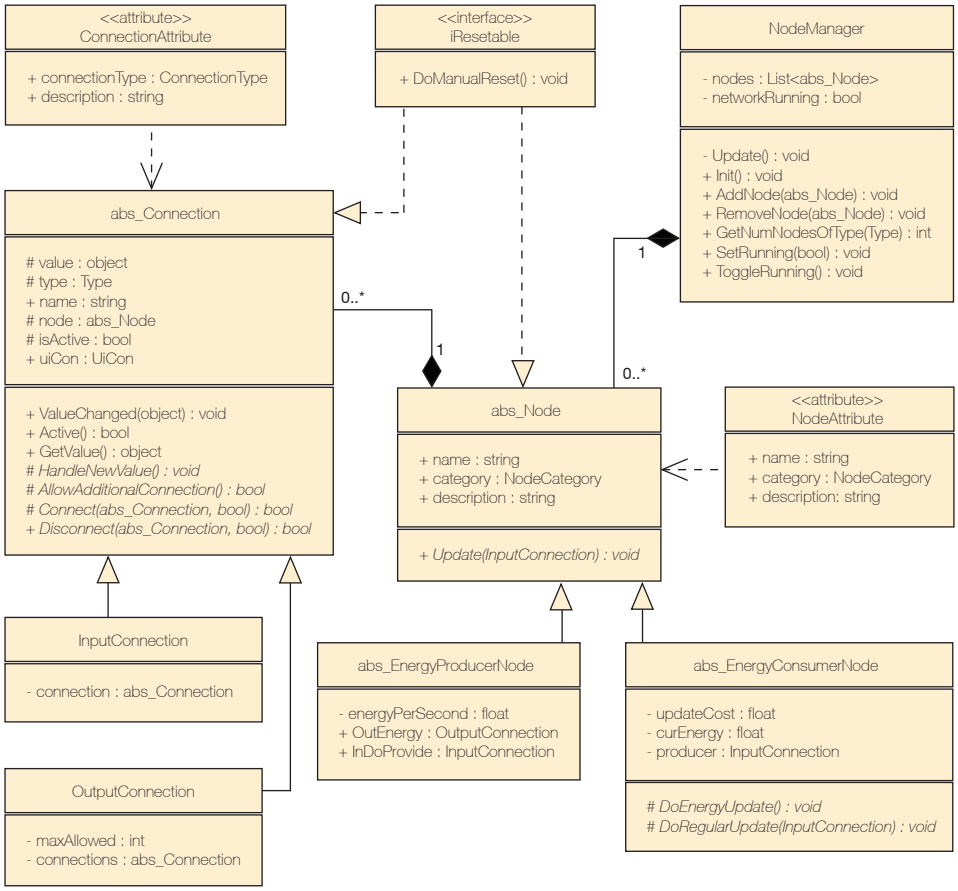


Figure 4.2: This class diagram shows the most important elements from the node system.

4.1.2 Node System

The node system is the very core of the project. It utilizes concepts of FBP that where described in section 2.2, though being rather classified as FBP-like system, as there are several concepts not implemented or adopted — section 4.1.2 describes these differences. The node system is built around a *singleton* implementation of a manager-class called `NodeManager` that keeps track of the active nodes, updates them and further handles resets of the node system, whenever the network is stopped. Figure 4.2 outlines important classes and their most important variables and functions, you can find the `NodeManager` class in the top right corner.

The very base for all nodes is the abstract class `abs_Node`, handling the registration at the `NodeManager` as well as storing data like the node's category (Energy, Robotics, Math,..). All existing nodes are directly derived from `abs_Node` or from one of the further specialist abstract classes

Program 4.1: The class `abs_EnergyConsumerNode` implementing the abstract method `UpdateNode` from the base class `abs_Node`.

```
1 public override void UpdateNode(InputConnection valueChanged)
2 {
3   if (valueChanged == this.EnergyProducer)
4   {
5     this._currentEnergy += ((Energy)this.EnergyProducer.GetValue()).
      amount;
6
7     if (this._currentEnergy >= this._costToUpdate)
8     {
9       this._currentEnergy -= this._costToUpdate;
10      this.DoEnergyUpdate();
11    }
12  }
13  else
14    this.DoRegularUpdate(valueChanged);
15 }
16 protected abstract void DoEnergyUpdate();
17 protected abstract void DoRegularUpdate(InputConnection valueChanged);
```

`abs_EnergyProducerNode` or `abs_EnergyConsumerNode`. Only energy cores implement the first one as they are the only nodes that produce energy while all Robotics nodes are derived from the second class. The class `abs_Node` also exposes the following important abstract method:

```
public abstract void UpdateNode(InputConnection valueChanged);
```

This method defines the functionality of the node. Whenever an `InputConnection` gets updated (whether from an active connection or by the user in the editor), `UpdateNode` is called, performing its functionality with the new value. Derived classes implementing this method have to check which input connection did change and then handle the new value of particular connection.

Program 4.1 shows `abs_EnergyConsumerNode` implementing `UpdateNode` to internally supervising its energy pool. If the input that changed is its dedicated input for energy, it updates the currently available energy and performs a special `EnergyUpdate`, whenever there is enough energy for. If it is not the energy input, it forwards it to another abstract method to handle all other inputs that are not energy-dependent i.e., changes on input values.

Establishing connections between nodes does work a bit differently compared to Morrison's system (see section 4.1.2) but there are also elements implemented that are inspired by him — like Initial Information Packets (IIPs). Whenever two nodes get connected to each other, an IIP is sent from the output to the input, establishing the connection and updating the

current value from the output to the input.

Differences to Classical FBP

While several concepts of Morrison's definition of classical FBP are incorporated into the prototype, there are also big differences. The goal was to create a system that fits rather than fully implementing classical FBP, thus classifying NO:LE:AP as a FBP-like system.

IPs and Connections: In classical FBP, there are dedicated Connections with bounded buffers handling IPs with a certain lifetime. NO:LE:AP does not incorporate these concepts as there are no large amounts of IPs, waiting to get processed by the targeted node, or deeper IP trees. Instead, connections are a mixture of connections and ports while input- and output-connections do directly communicate with each other. Outputs immediately update connected inputs, which again triggers the update method of the corresponding node. This makes IPs with a certain lifetime redundant. Instead, all connections do have a well-defined data type they are allowed to consume and all connections with differing data types are immediately refused — more on data types in section 4.1.2.

Multiple Connections: Classical FBP does not allow multiple input-connections to the same output-connection, because Morrison [24] sees data objects behave like real world objects, thus they cannot get magically duplicated. In our implementation, similar to most of other fbp-like ones, this is allowed though. While inputs can establish only one concurrent connection, outputs can be connected to more than one input. Still, the current implementation does allow to restrict the maximum amount of connections, and while the default setting does not define a maximum, there are some nodes that do use this restriction. An example are energy nodes, as they do allow only a certain number of energy consumers connected to their energy-output at the same time.

Asynchronous Processes: While classic FBP is highly asynchronous as every process is running in its own thread, NO:LE:AP nodes are not automatically asynchronous, as most of the nodes do not have very performance-heavy calculations. Unity3D has its own system handling asynchronous tasks called *Coroutines*. As it is a classical game engine, there is an internal update-loop and coroutines do allow methods to pause their execution, continue the game's update-loop and return to the method in the next update cycle by calling the `yield return` statement. This is a very stable and deeply integrated system but also costs some performance to allocate a new asynchronous coroutine. Thus, nodes do work synchronous by default and while this is sufficient for most of the logic in the existing node's update

methods, it is highly recommended to utilize coroutines for heavier calculations.

Data Types

Every connection defines the exact type of data it can send or receive. Connections handling different data types cannot be connected to each other. Currently, following types do exist:

Standard Types like `Int`, `Float` or `Bool` are essential for lots of nodes and are all supported by default.

Vector2 is a default type from Unity and represents a two-dimensional vector i.e., two floats. NO:LE:AP's nodes mainly use them for positional calculations.

Selectable defines a Gameobject in the current world and its `SelectionType`, which is an *Enum*. Currently, there are three `SelectionTypes` available, `All`, `ARE` and `Moveable`. It is used for filtering the Gameobjects in the current world for selection-lists. An example are Roboticsnodes that directly control A.R.E.s. They filter for the `SelectableType` `ARE` to restrict players to only select an A.R.E. to be manipulated.

Energy has no further purpose than to define connections that are used for transferring energy between energyproducer and -consumer.

Equation is a special type only used for logical nodes like the *Float Comparator* at the moment and contains logical conditions like `equal`, `greater` or `lesser or equal`.

AllowedKeys defines a set of letters representing the keys on a keyboard that can be used for player input — solely used by the special node *Key Input* at the moment.

C# Attributes and Reflection

To keep the node system as flexible as possible, a feature of C# called *Attributes* was utilized. It is a way of storing additional information called metadata which can be retrieved during runtime with *Reflection*. On this way new nodes can be easily created without maintaining lists of available nodes. There are two custom attributes, `NodeAttribute` and `ConnectionAttribute`. The first one stores the name of the node, its category and a description, the second one defines the connection type and a description.

Program 4.2 shows the full source-code for the node *Logic Negate*. There are just a few lines of code to define the whole node with a total of three attributes for the node itself and its two connections. The method `UpdateNode` implements the node's logic, in this case, it simply inverts incoming boolean values.

Program 4.2: The full source-code for the node *Logic Negate* which outputs an inverted boolean input value. Mind the German text, as the game is currently only localized for German players.

```
1 [NodeAttribute("Logik - Negieren", NodeCategories.Logik, "Dreht den
      einkommenden logischen Wert um. Aus WAHR wird FALSCH und umgekehrt."
      )]
2 public class LogicNegateNode : abs_Node
3 {
4   [ConnectionAttribute(abs_Connection.ConnectionType.Input, "Der
      logische Eingabewert")]
5   public InputConnection InLogic { get; set; }
6
7   [ConnectionAttribute(abs_Connection.ConnectionType.Output, "Negierte
      Ausgabe")]
8   public OutputConnection OutLogicNegated { get; set; }
9
10  public LogicNegateNode() : base(NodeCategories.Logik)
11  {
12    this.OutLogicNegated = new OutputConnection("Negiert", this, false);
13    this.OutLogicNegated.Activate();
14
15    this.InLogic = new InputConnection("Logik", this, true);
16    this.InLogic.Activate();
17  }
18
19  public override void UpdateNode(InputConnection valueChanged)
20  {
21    this.OutLogicNegated.ValueChanged(!(bool)this.InLogic.GetValue());
22  }
23 }
```

4.1.3 Network Editor

While the node system previously described does work on its own, it has no visual representation yet. To allow the player to create behavior-networks without to program them, the network editor is the graphical representation of the node system.

Whenever a world is selected, only a few nodes are unlocked to complete given tasks. Reflection is used to find all classes implementing `NodeAttribute` which are then filtered regarding their availability in this world. The list of available nodes in the network editor is sorted regarding their category but as `NodeAttribute` contains the node's category, only for the currently selected category, list-entries are created.

The core class for managing available nodes and creating new nodes is called `UINodeManager`. Whenever an entry from the list of available nodes is dragged into the network window, `UINodeManager` is called to create a new graphical representation of this node. This system is very dynamic as

Program 4.3: Part of the source-code of `UINodeManager` creating a new node. Reflection is used to get all properties with `ConnectionAttribute` to create the node's input- and output-connections.

```
1 foreach (PropertyInfo property in nodeData.Type.GetProperties())
2 {
3   var atts = property.GetCustomAttributes(true);
4
5   foreach (object att in atts)
6   {
7     ConnectionAttribute con = att as ConnectionAttribute;
8
9     if (con != null)
10    {
11      // create graphical connection here
12    }
13  }
14 }
```

there is only a single base for all nodes. The node's logic gets defined by the following line of code:

```
node.NodeLogic = (abs_Node)Activator.CreateInstance(nodeData.Type);
```

Activator is another feature of C# that allows to create instances of given types — in this case the type of node dragged to the network window. To create graphical representations for all connections, reflection is utilized again. Program 4.3 shows a code snippet that obtains all properties with `ConnectionAttribute` in the currently selected node-type. For each found attribute, a connection is created. As they always keep the latest value they received and thus are never empty, they get a default value depending on the data type they have defined. Finally, the labels of the node and its connections are assigned and the node is ready to be used.

To connect two nodes with each other, the user can perform a drag and drop gesture from one connection to another. A line follows the user's gesture and if the drop gesture is valid, the line stays after the gesture, presenting the active connection between the input and the output. The gesture is valid when it started at an input and ended at an output or vice versa, the data type is the same and there are not already too many active connections.

To be able to draw lines in Unity3D's UI system, a custom class `LineRenderer` class was implemented. All connections keep a reference on all their active connections not only redraw them, whenever a node gets dragged around but also the know, which connections have to be removed when a node gets deleted. As the player can delete nodes and connection-lines as well, they both implement a special interface called `IDeletableElement`.

Whenever a node gets deleted, all lines get removed as well and all connected nodes get informed about loosing an active connection. When the user deletes a line, both nodes that where connected through this line get updated.

Another feature of the prototype utilizing attributes are the tooltips that where introduced after the early playtests (see section 3.3). Whenever the user hovers over a node or a connection, a tooltip appears after a short amount of time, revealing further information. These descriptions are also stored within `Node-` and `ConnectionAttribute`.

4.1.4 Worlds and Tasks

To easily add new worlds with several tasks to be solved, a complex world- and task system was implemented. All worlds are defined by a *Struct* called `World` containing all information needed including all hints, hidden hints, descriptions and all required tasks. A singleton class called `WorldSelectionManager` keeps track of all available worlds and is also responsible for loading selected and unloading finished ones. Tasks are defined with following structure:

```
new string[][]
{
  new string[] { "TaskTouchTarget", "A.R.E. 1", "Schalter 1" },
  new string[] { "TaskTouchTarget", "A.R.E. 2", "Schalter 2" }
}
```

This is a two-dimensional string-array with each entry representing a task. While the first string always defines the type of task, all other entries do vary. In given code example above, two `TaskTouchTarget` instances are created, with the first requiring *A.R.E. 1* to touch the game entity *Schalter 1* and *A.R.E. 2* to touch *Schalter 2*.

The core class for managing tasks is a singleton implementation called `TaskManager`. It generates the tasks for the selected world and keeps track of their status. Tasks are generated through the method `GenerateTasksFromRaw` which parses the two-dimensional array and instantiates a task depending on its type. All tasks derive from an abstract base class called `abs_Task` which requires deriving classes to implement two abstract methods:

```
protected abstract bool CheckCompletionInherited();
protected abstract void ResetTaskInherited();
```

`TaskManager` regularly checks all active tasks for their completion by calling `CheckCompletion`. Whenever a running network gets stopped, the worlds resets and `TaskManager` calls `ResetTask` to set all tasks to their default. As soon as all tasks are completed simultaneously, the world is finished and the game switches to the *summary screen*.

Program 4.4: Example program for utilizing Unity's `WWW` class to send requests to the server.

```
1 private static IEnumerator LoginUser(string user, string pw)
2 {
3   // create request
4   WWWForm f = new WWWForm();
5   f.AddField("name", user);
6   f.AddField("pw", Instance.EncryptString(pw));
7
8   WWW w = new WWW("http://url-to-rest-api.com/login_auth", f);
9
10  yield return w;
11
12  if (w.error == null)
13  {
14    // everything is fine, handle request response
15  }
16 }
```

4.1.5 Persistency and Metrics

All communication with the server is done through a singleton class called `WebService`. Unity offers a class called `WWW` that is utilized to send *http requests* to the server's REST API. Data is sent as a POST request with *content-type: form-data* which can be achieved by creating a new instance of Unity's `WWWForm` class. This object can be then filled with key-value pairs and added to a http request (see program 4.4). These requests are asynchronous, thus happen in the background or during a loading animation. Whenever a player starts NO:LE:AP, following requests are sent:

- During the initial loading screen, a check for a working internet connection is done. If the server is reachable, the first request is a check for the application's version number. If the client has a different version than the server, the client is not capable of log-in properly and the game immediately switches to an error screen telling the player to download the latest version.
- Entering the log-in screen, the player can now log-in with existing username and password or register a new user.
  - When the player is already registered, a log-in request is sent, checking if the entered username does exist and if the password does match with the password stored on the server. All errors are displayed on the bottom telling the player what is wrong with the log-in attempt.
  - The player can also register a new user at the *register screen*. A local check is done for all fields being filled out correctly and a

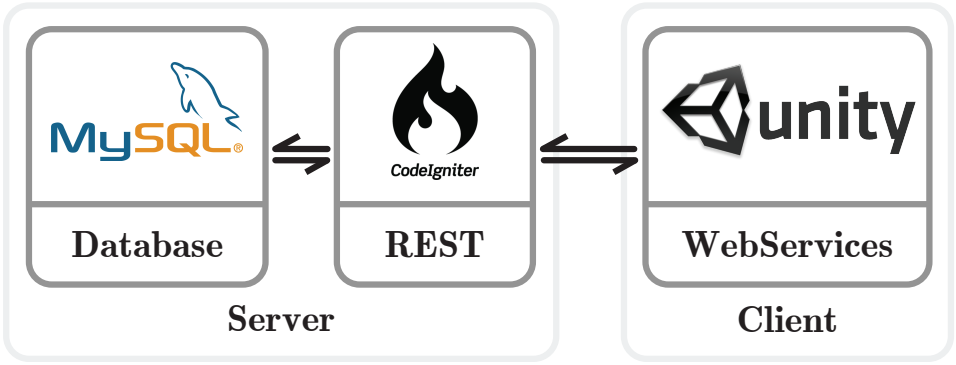


Figure 4.3: This figure shows the used software and how they communicate with each other.

registration request is sent to the server. If the username or the e-mail does not already exist, the user is registered successfully and the player can now log-in.

- When the log-in is successful, a `fetchUserData` request is sent, gathering all information about the active player from the server. This contains the available points to spend, all unlocked levels and the obtainable points remaining.
- Unlocking a new world does also trigger a request, checking for sufficient points and updating the database.
- Whenever a player finishes a world, the player metrics are synchronized to the server, e.g., time taken, components used or restarts. If there where obtainable points left, the player's available points get updated too.
- Finally, all aborted games get tracked too — see section 3.3 for an explanation.

4.2 Server

While the server is an essential part of the project and is used to keep track of the player's progression, his data and all the player metrics, there is clearly no focus on the technical implementation in this thesis. This section gives a brief overview of the server's software and implementation.

4.2.1 Software Stack

The goal of the server-side implementation was to provide a solid API to the client and a way of securely storing the collected data (see figure 4.3). Following software was used to achieve these goals.

MySQL

MySQL describes itself as "the world's most popular open source database" and is a relational database management system. It exists since more than 20 years, is owned and developed by *Oracle* and is a very mature software. Lots of very prominent projects are using MySQL like Wordpress, Facebook, TYPO3 or Youtube.

CodeIgniter

CodeIgniter is a light-weight PHP framework for fully-featured web applications. While it is open source, very easy to use and it has a large community, it is lacking a very important feature as there is no native support for REST services. Fortunately, there is a very mature open source implementation of a CodeIgniter REST server which can be found on Github[2].

4.2.2 Implementation

After setting up a server and installing MySQL, CodeIgniter and the REST server, a few initial configurations have to be done. CodeIgniter makes this a fairly easy task as it provides several configuration files including one for our REST API and for database connections.

Database Structure

The next step is to create the structure of the database. Currently, following tables exist:

`users:` This table contains all the information about the currently registered users. This is mainly the data gathered on the registration screen i.e., the username, an encrypted version of the user's password and the demographic data. There is also a primary key called `id`, a unique auto-incrementing integer used for dependencies between tables. Finally, the registration date and the current available points are stored as well. The current table with the first three entries is shown in figure 4.4(a).

`worlds_unlocked:` To keep track of the worlds every player has unlocked, the table `worlds_unlocked` adds an entry for each unlock transaction. The columns of this table are the unique id of the user, the unique id of the unlocked world, the date of the transaction and the unique id of the best try — which is another table called `games_completed`. This enables the database to join both tables together and easily get the best try and thus the remaining points the player can obtain from this world.

[2]The link to the repository is https://github.com/chriskacerguis/codeigniter-restserver.

id	name	pw	mail	age	sex	highest_edu	current_ac	computer	program	math	riddles	new_tech	games	registration	xp
30	[redacted]	[redacted]	[redacted]	21	1	2	1	1	5	5	2	2	1	2016-07-14	75
27	[redacted]	[redacted]	[redacted]	26	1	7	3	1	2	2	3	1	1	2016-07-11	130
28	[redacted]	[redacted]	[redacted]	25	1	3	3	3	5	2	3	3	1	2016-07-11	50

(a)

id	user_id	world_id	best_try_id	date_bought
82	27	TriggerTimingMedium	*NULL*	2016-07-11 13:40:22
77	27	TutMoveOne	80	2016-07-11 13:06:40
78	27	SelektRotateEasy	81	2016-07-11 13:14:34

(b)

id	user	level	restarts	hints	total_time_secs	level_started
122	25	SelektRotateEasy	0	0	6	2016-08-24 15:55:29
123	25	TriggerTimingMedium	3	1	91	2016-08-24 23:38:21
120	25	ControlsStromlMedium	0	0	5	2016-08-24 15:37:29

(c)

id	user	level	points	over_optimum	restarts	hints	total_time_secs	level_started
94	29	TriggerTimingMedium	100	0	0	0	34	2016-07-12 19:16:17
95	29	DanceMedium2Robots	90	0	2	0	155	2016-07-12 19:17:20
93	29	TriggerTimingMedium	80	2	0	0	175	2016-07-12 19:12:00

(d)

Figure 4.4: The four tables of the current database including their first three entries: users (a), worlds_unlocked (b), games_aborted (c), games_completed (d).

`games_aborted:` This table, shown in figure 4.4(c)), contains all worlds that where aborted by the users, whether through clicking the *Leave World* button or through directly closing the game. All available information are uploaded as well, like the network restarts, unlocked hidden hints or the total time taken until the world was aborted.

`games_completed:` Whenever a world is completed, all information regarding this try i.e., the player metrics of this world, are uploaded. The table contains the unique user id, the unique world id and the player metrics like network restarts, unlocked hidden hints or the total time taken (see figure 4.4(d).

CodeIgniter REST Server

Implementing the REST API is also fairly straight-forward. Deriving from a class called *REST_Controller* tells the framework, that this class defines methods reachable through our API. All methods following a certain naming convention — *methodname_requesttype* — are automatically callable. An example is *login_auth_post* where *login_auth* is the name of the method and *post* defines the http method of the request (e.g., post, put, get or delete). This particular method is called, whenever a user tries to login, therefor a username and a password has to be added to the request. Post-parameters can be obtained by calling

```
$this->post("key-name");
```

which returns either *NULL* if the key does not exist or the corresponding value. As the example method has to check if the user exists and the password does match, a query to our MySQL database has to be sent. This is done by calling CodeIgniter's native method *query*, thus checking if the given username does exist in our database is the following single line of code

```
$this->db->query("SELECT * FROM users WHERE name = '" . $this->post("
    name") . "'");
```

which returns an array with the response of the query. All http requests do result in a response which is done by calling following method

```
$this->set_response(['status' => 'success'], REST_Controller::HTTP_OK);
```

The first parameter is an array of key-value-pairs that can be freely defined and contains data the client requested. The second parameter defines the http-status in the http-header.

Chapter 5

Evaluation

As soon as all systems explained in chapter 3 and 4 did work properly, gathering data was a fairly easy task. The user's relevant demographic data is collected during the registration process and their game's performance is automatically synchronized too.

To be capable of optimally working with the gathered data, a small software pipeline was created. A great and free solution to work with lots of numerical data is *Google Spreadsheet* which also supports writing cell-functions, similar to *Excel*. To get the data out of the MySQL database into a table and its sheets, another software is used called *Google Apps Script*. This allows to write complex *Java* functions that can be embedded into Google's other apps, like Spreadsheet. To connect and send queries to the database, *Java Database Connectivity (JDBC)* can be utilized within Apps Script. The final implemented function connects to the database, collects the existing data and automatically adds it to the spreadsheet. All calculations within the spreadsheet are written with functions, thus all data-updates from the function automatically result in updated values in the sheets. To not only present this data in plain numbers, *Adobe Illustrator* was utilized to process them into visually appealing diagrams.

At first, this chapter presents the gathered data in an objective way in section 5.1. Finally these metrics get analyzed and interpreted in section 5.2 to find out if there is material to support the stated theory in chapter 1.

5.1 Results

As of the date this thesis is written, a second wave of distribution is completed (with the early closed playtests from section 3.3 counting as first). This results in following records in the four tables of the database (which where described in section 4.2.2):

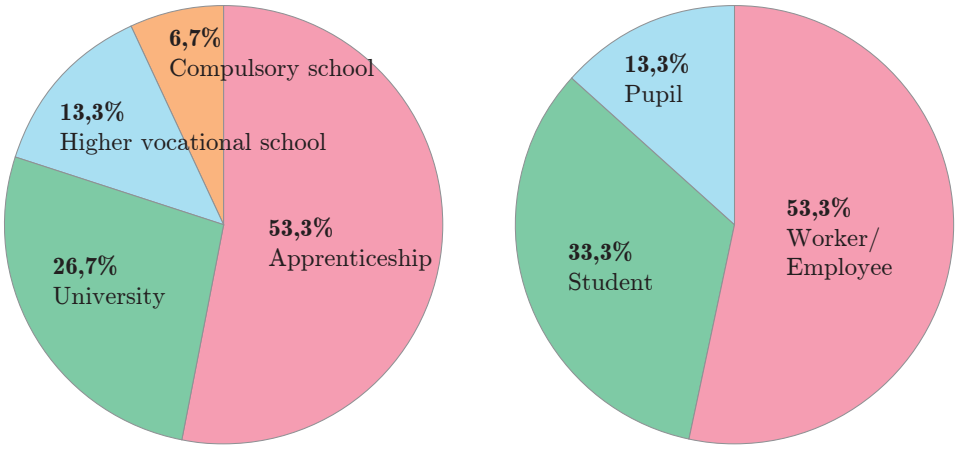


Figure 5.1: The left pie charts shows the user's highest completed education, the right one their current activity.

- 16 entries in `users`,
- 97 in `worlds_unlocked`,
- 100 in `games_completed`,
- and 29 in `games_aborted`.

After the first playtest, the database was completely cleared, thus all these entries are from the current version. While the amount of collected data is good enough for an evaluation, there could be of course much more due to the nature of automated collection. More actual players would have directly resulted into more data, but this second wave was still distributed in a controlled way with the focus on a broad variety of differently educated people with different interests. The idea behind such a restricted distribution was to get enough data for an analysis while still maintaining a large group of people not knowing NO:LE:AP to not distort the results of a possible third wave of distribution after further updates to the application — see chapter 6 for problems that occurred and an eventual outlook of this project.

5.1.1 Demographic Data

The 16 registered users consist of 14 male and 2 female subjects, their age ranging from 21 to 40 with an average of 27.4 years. Figure 5.1 shows their highest completed education and their current activity. More than half of them completed an apprenticeship and are currently working (which does not imply that these are the same persons). The rest is still receiving some sort of education, as there are 33.3% students and 13.3% pupils. With this fairly homogenous group regarding education and activity as a solid basis, the more interesting part is the demographic data as this will be used to

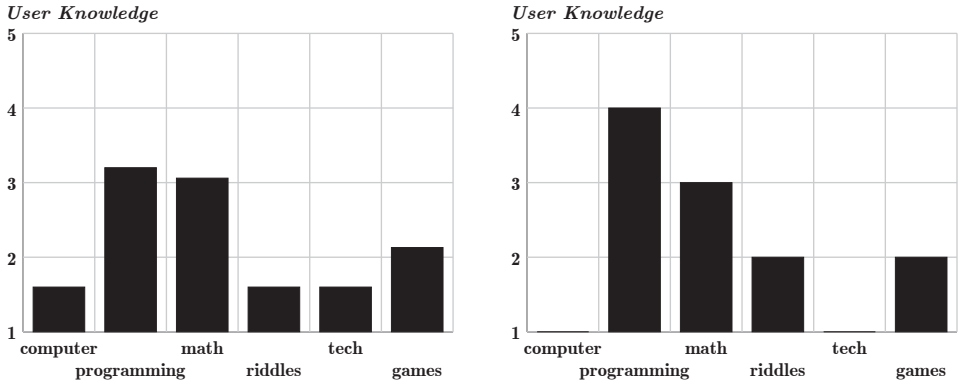


Figure 5.2: These figures show the calculated average and median values for the demographic questions.

classify the user later on. During the registration process in the *registration screen*, the user has to answer six questions with one of five available answers ranging from *applies to me a lot* to *applies to me in no way*. All questions are about the interest and the amount of time spent regarding a certain area, which are *computers* (in general), *programming*, *math*, *riddles*, *technology* and *games*.

In figure 5.2, two column graphs are shown. The left one shows the average answers, with the vertical axes presenting the possible answers (with 1 as *applies to me a lot*) and the horizontal axes the six questions. While the interest in computers, riddles and technology is very high and games are still interesting to them, programming and math are beyond average. The median values shown in the right column graph makes this even more clear as programming has a median value of only 4 (which resembles to an *applies to me rather not*) while their interest in computers and technology gets even stronger.

5.1.2 Played Worlds

Overall, the 16 users started 129 worlds in total, which then ran for more than 625 minutes (until they got completed or aborted). They also unlocked and even completed several worlds, is shown in figure 5.3. The vertical axis is percentage from zero to hundred and the horizontal axis is split in ten columns representing the amount of available levels in the current version of NO:LE:AP (which is exactly ten). The darker line represents the percentage of *total users unlocked given amount of worlds* i.e., 100% of all users unlocked at least one world which is steadily decreasing down to 40% who unlocked all 10 worlds. The second and brighter line is the percentage of *total users completed given amount of worlds* to the users who unlocked it. For example, in column 5 do both lines overlap which means that the same amount of users

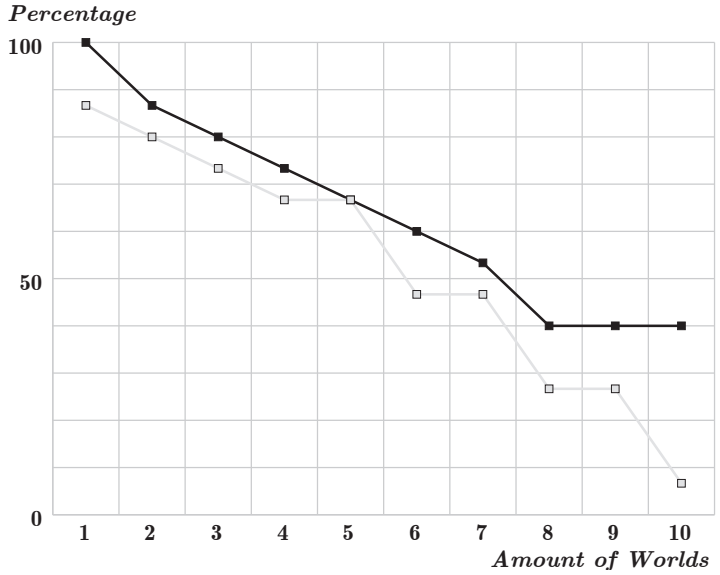


Figure 5.3: This figure compares the amount of unlocked worlds (the dark line) to the amount of completed worlds (the bright line) in percentage of the total users.

that unlocked five worlds also managed to accomplish five worlds (which is roughly 65% of all registered users). Another interesting comparison is the first column where 100% unlocked at least one world but not all of them accomplished this world. This is especially odd as the first world is part of the tutorial which should be no problem at all — an explanation can be found in section 6.2.

5.2 Interpretation

5.2.1 Analysis of the Progression System

The way the progression system currently works is exhaustively described in section 3.2. While there went a fair amount of time and thoughts into this system, it was not really clear if it also works as intended. Currently, users earn a certain amount of points whenever they complete a world, depending on how well they did and how many penalties they got. Unlocking worlds costs points, but completing a world in an ideal way (i.e., with 100 points) gives the users more points than unlocking new worlds costs. This leads to more and more unused points the users can spend to unlock more than one world at a time if they should be unable to finish a just unlocked world.

Figure 5.3 shows that the users did indeed utilize this system. During the first several worlds, there are constantly users not completing unlocked

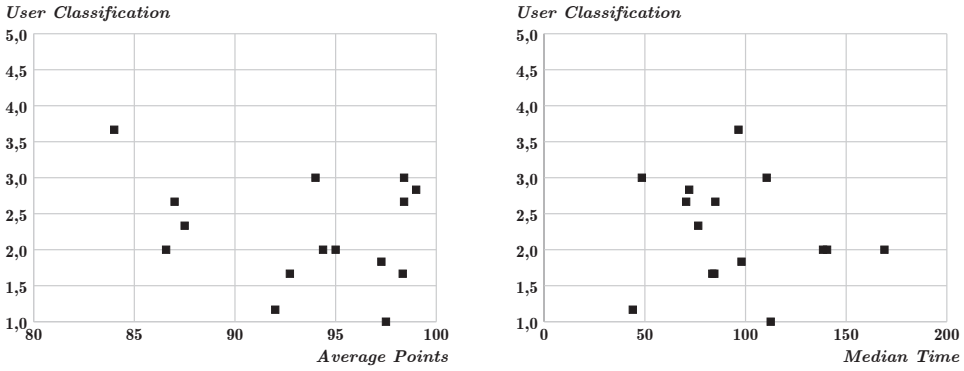


Figure 5.4: The left figure shows the relation of the user's skill to the average points he achieved on the first successful completion, whereas the right one the user's skill in relation to the median time taken.

worlds and the amount of total users progressing further steadily decreases. There could be several explanations for these drop-outs like they simply stopped playing because they do not like the game. Beginning with the second half of the worlds, more and more of them get unlocked while less worlds getting actually completed. Users already gathered enough points to unlock more than one world at once, whenever they did not manage to complete one, resulting in 40% of all users unlocking *all ten worlds* while only less then 10% also managed to complete all of them.

5.2.2 Supporting the Thesis's Theory

Looking at the demographic data in figure 5.2, the actual interests and spent time of the users can be described as a good distribution in terms of using them to validate the theory of this thesis. Computers and technology in general are of very high interest, followed by riddles and games. Math and especially programming are areas they are much less into. Given these values, the average user of our second wave of testing can be described as a *technology-affine subject who likes solving logical (and digital) problems while not being used to think in a programmatic way.*

Classifying the Users

The thesis goal is to find out if visual node-based editors can help to lower the entry level to areas like programming or math and thus can be described as beginner-friendly and a tool that can be utilized for educational purposes. The average user can be indeed considered as beginner and thus being part of the target group, looking at the just stated definition and only taking the areas the average user is less potent into account. But all six questioned areas are equally important, as bad self-estimations in all of them could

influence the outcome. A user who is very interested into math but never used a computer before is also handicapped in a certain way and could at least perform equally bad as someone with the opposite skills. Thus, all six areas will be taken into account equally. Otherwise, the highest completed education and the current activity should not directly influence the classification of the users. Highly educated ones do not necessarily surpass others automatically in the questioned areas, as the focus of their education could be on completely unrelated areas.

These thoughts result into a user-classification by *calculating the average of the six areas of interest*. A user with a low value can thus be called a professional while a high value marks a beginner. While this seems like a reasonable classification, figure 5.2 should be still considered as there are large differences between the interests resulting in fairly high average classifications while a large group of the users are indeed programming and math beginners.

Comparing the User Performance

Figure 5.4 utilizes just calculated classification to compare the performance of the users in two graphs with the user-classification in both vertical axis. Looking at the vertical distributions, it seems that there are not that many beginners as most of the users classify themselves above the mean value. Again, the large differences between the interests have to be considered and filtering them for just programming and math would result in way more users classified as beginners. The left graph shows the average earned points on the first successful completion. While users can play worlds multiple times to gain all available points, *only the first try is the one that counts for this graph*, as with every other try, users already know the world and automatically perform much better. Furthermore, to compensate any distortions of the earned points by the system's flaw described in section 3.3, *an average of the penalty-points of the user's aborted worlds is additionally subtracted*. The calculated values range from 84 points up to 98.

The second graph compares the user-classification with the *median time taken*. At first, the average value was calculated too, but there where some extremely long times measured with values up to 45 minutes which is interpreted as *absence of the user* by the author and thus would distort the results. Similar to the first graph, the times of the aborted worlds are incorporated as well. In this graph, lower values are better results as they point to faster completion times. Also mind that time was never indicated as a measuring factor in NO:LE:AP and high values do not result in penalties, thus the users did not play fast nor felt the pressure of time that could influence the outcome negatively. Using time they took to complete worlds is still a valid way of comparing their performance with each other.

The most interesting part in figure 5.4 is the bottom right quarter of

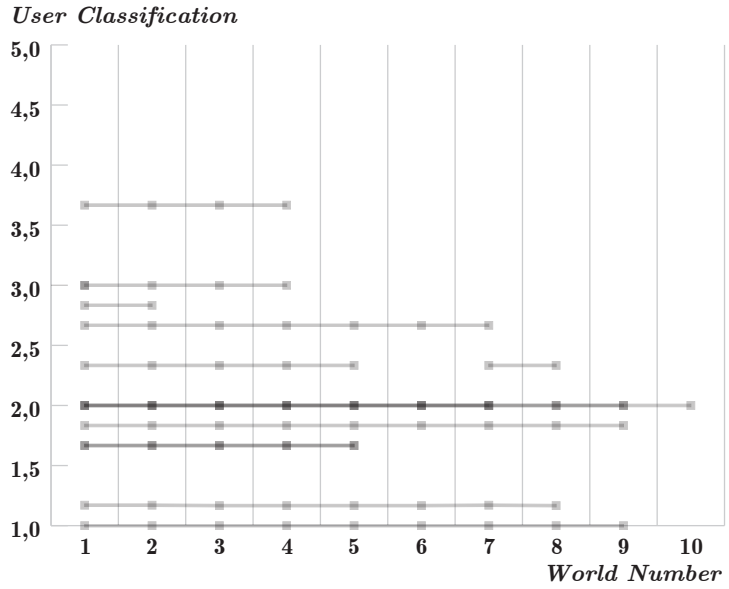


Figure 5.5: This figure shows the accomplished worlds in detail. Mind the slightly darker lines indicating accomplished worlds of users with identical classification i.e., have overlapping lines.

the left graph and the bottom left quarter of the right graph. There is a large majority of the users performing very well with an average number of achieved points way above 90 and a median time of less than 100 seconds per world. Looking at the overall good performance already is a success regarding the network editor, the systems in place and NO:LE:AP in general. But taking the distribution of the user-classification into account really indicates that there is material that could support the theory of the thesis. As there is clearly shown, *the best as well as the fastest ratings are evenly distributed among users classified between one and three.* This means that not only the users that classified themselves as very good in all six areas, including programming and math, did very well but also users with an average self-estimated classification down to three. Looking at these users in detail further supports this interpretation as they all have good values regarding computer, technology or games and very low self-estimations in the fields of programming or math. Still, they perform equally to the better classified users which certainly hints to the network editor helping them and lowering the entry level to given areas for them. *Users with much less experience in fields like programming or math do perform as good as users with knowledge in these fields while using NO:LE:AP's network editor.* Another positive sign comes up when looking at the performance of the users when replaying worlds as they all perform much better in later tries which could also indicate a certain aspect of learning progress.

One eventual pitfall came up which could probably distort this interpretation. As figure 5.3 showed, there where users constantly dropping out. This could lead to users being placed next to the best players by completing only very easy worlds with high points. To find out, which users did drop out at which stage and if this would significantly influence the given interpretation, figure 5.5 is added. With the vertical axis again representing the user-classification, the columns of the horizontal axis represent the ten available worlds. While the horizontal axis in figure 5.3 represented the total amount of worlds, the horizontal axis of this graph represents the actual worlds. For example, looking at the user with a classification of approximately 2.4, the graph shows that he accomplished the first few worlds then left out world six and finally again accomplished worlds seven and eight. This graph also shows that there is not really a reason to fear this believed pitfall as most of the users that did really well are also the ones that accomplished the most worlds, regardless of the classification.

Chapter 6

Conclusion

This last chapter gives a final overview of the project's goals and what was actually achieved. Furthermore, eventual problems that came up during the development process are summarized too, concluding with an outlook of eventual next steps.

6.1 Summary

This thesis tried to find out, if a visual flow-based programming editor can help to lower the entry level to areas like programming or math. To achieve this goal, a solid foundation of knowledge about given topic had to be accumulated it first. Multiple concepts where developed on this basis, shaping into a prototypical approach of a learning game called NO:LE:AP with the goal to gather data from its users to support the theory of this thesis. After the systems and concepts that made it into this prototype where presented, a detailed insight into the technical aspects and the actual implementation was given. First tests where conducted, resulting in feedback to the prototype and sufficient data for a first evaluation.

After presenting given data, it was finally analyzed and interpreted to find valuable metrics supporting the thesis's theory. While the total amount of 16 registered users during the second wave of distribution is not enough to really qualify the collected data and the resulting interpretation as convincing, some important flaws where detected and corrected and the overall conclusion is very promising. While more work and especially more data is needed for a final conclusion, the current status does at least not disprove the thesis's theory.

6.2 Problems

As the project grew over time, the amount of known bugs also grew simultaneously, resulting in a fairly long list of minor and major problems that

did not make it into the current version. Several test users further reported problems with certain screen resolutions (and display aspect ratios) or even minor bugs with some game entities, meaning that this list is still growing.

A severe problem within NO:LE:AP, that has to be fixed immediately, came up during the analysis of figure 5.3 in section 5.1. It was very confusing that there where users that did not even finish the very first world. Whenever a newly registered users starts NO:LE:AP for the first time, the tutorial automatically starts and by following its instructions step-by-step, everything gets explained to solve the first world without any problems. But still, there where users that did not complete any worlds. As all test users of the second wave are personally known to the author, the ones that did not complete the tutorial could be questioned about their reasons. The solution was fairly simple but also showed that there still has to be done lots of optimization. These users told the author, that the tutorial was too long for them with too many pages filled with too much text. They simply did not want to read through the whole tutorial but then they also where not capable of finishing the first world as they did not know what to do.

6.3 Outlook

NO:LE:AP became a huge project during its development, thus there are still lots of things to add and optimize. But as the final outcome at the end of this thesis is thoroughly positive, the motivation is very high to continue developing NO:LE:AP. For a third, even bigger distribution wave than the previous ones, usability will be an important topic. There have to be more features on the project's website like a way of resetting the user password when forgotten, as this already happened in the second wave with a fairly small amount of users — fortunately all test users where personally known to the author, thus the password reset was done manually.

An even larger variety of worlds is also intended, as more worlds mean more spare points to skip too hard worlds. More worlds does also mean more time the users can spend with NO:LE:AP which results in more data. Thanks to the very flexible world and task system, this should not be a problem.

But one of the first things that has to be done is a further analysis of the collected data, how this data will be interpreted in the future and which metrics could be also interesting for the project's evaluation. As this is the core of the whole project, it has to be a fully satisfying process and the more concrete the results will get in the future, the more NO:LE:AP can be seen as success.

Appendix A

Contents of the CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Thesis

Pfad: /

Stuntner_Benjamin_2016.pdf Master thesis with instructions (entire document)

A.2 Online Sources

Pfad: /online-sources/pdf

comp-based-se.pdf	Component-based software engineering, Wikipedia
construct2.pdf	Construct 2
deutsch-limit.pdf	Deutsch limit, Wikipedia
fbp-github.pdf	Flow-Based Programming, Terminology
gpss.pdf	GPSS, Wikipedia
ue4-blueprints.pdf	Using Blueprints in Unreal Engine 4, Thomas Ingham
json.pdf	JSON, Wikipedia
fbp-morrison.pdf	Flow-Based Programming, John Paul Morrison
noflo-vs-fbp.pdf	NoFlo vs. “Classical” FBP, John Paul Morrison
noflo-jekyll-github.pdf	noflo-jekyll
prog-paradigm.pdf	Programming paradigm, Wikipedia

Pfad: /online-sources/others

azzolini-dfd.ppt	Introduction to Systems Engineering Practices, John Azzolini

A.3 Images

Pfad: /images

*.pdf	PDF files
*.psd	PSD files
*.png	PNG files
*.jpg	JPG files

A.4 NO:LE:AP

Pfad: /client/source

*	Source-Code NO:LE:AP v1.0

Pfad: /client/build

*	Build NO:LE:AP v1.0

Pfad: /server/rest-api

*	Source-Code Server v1.0

Pfad: /server/website

*	Source-Code Website

References

Literature

[1] Andrew Begel. *LogoBlocks: A Graphical Programming Language for Interacting with the World*. Tech. rep. MIT Media Laboratory, 1996. URL: http://research.microsoft.com/en-us/um/people/abegel/mit/begel-aup.pdf (cit. on pp. 7, 8, 11, 12).

[2] Marat Boshernitsan and Michael Sean Downes. *Visual programming languages: a survey*. Tech. rep. UCB/CSD-04-1368. Berkeley, CA: Univ. of California, Computer Science Division (EECS), 2004. URL: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-04-1368.pdf (cit. on pp. 4, 11).

[3] Timothy B. Brown. *Completeness of a Visual Computation Model*. Tech. rep. WUCS-93-53. Washington Univ., St. Louis: All Computer Science and Engineering Research, 1993. URL: http://openscholarship.wustl.edu/cse_research/548 (cit. on pp. 3, 5).

[4] Margaret Burnett et al. "Forms/3: A First-order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm". *Journal of Functional Programming* 11.2 (Mar. 2001), pp. 155–206 (cit. on p. 6).

[5] Shi-Kuo Chang, ed. *Principles of Visual Programming Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990 (cit. on p. 3).

[6] Xin Chen et al. "A Model of Component-Based Programming". In: *Proceedings of the Intl. Symposium on Fundamentals of Software Engineering*. Ed. by Farhad Arbab and Marjan Sirjani. Tehran, Iran: Springer Berlin Heidelberg, Apr. 2007, pp. 191–206 (cit. on p. 7).

[7] Melvin E. Conway. "Design of a Separable Transition-diagram Compiler". *Communications of the ACM* 6.7 (July 1963), pp. 396–408 (cit. on p. 13).

[8] M. Erwig and B. Meyer. "Heterogeneous Visual Languages-integrating Visual and Textual Programming". In: *Proceedings of the 11th International IEEE Symposium on Visual Languages*. VL '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 318– (cit. on p. 4).

[9] Ephraim P. Gilnert and Steven L. Tanimoto. “Pict: An Interactive Graphical Programming Environment”. *Computer* 17.11 (Nov. 1984), pp. 7–25 (cit. on p. 10).

[10] Masahito Hirakawa and Tadao Ichikawa. “Visual Language Studies - A Perspective”. *Software - Concepts and Tools* 15.2 (1994), pp. 61–67 (cit. on p. 10).

[11] Jill H. Larkin and Herbert A. Simon. “Why a Diagram is (Sometimes) Worth Ten Thousand Words”. *Cognitive Science* 11.1 (1987), pp. 65–100 (cit. on p. 10).

[12] Tim Menzies. “Evaluation Issues for Visual Programming Languages”. In: *Handbook of Software Engineering and Knowledge Engineering*. Ed. by Shi-Kuo Chang. Vol. 2. Canada: World Scientific Publishing Company, 2002, pp. 93–101 (cit. on pp. 4, 10).

[13] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Paramount, CA: CreateSpace, 2010 (cit. on p. 14).

[14] K. N. Whitley and Alan F. Blackwell. “Visual Programming: The Outlook from Academia and Industry”. In: *Proceedings of the Seventh Workshop on Empirical Studies of Programmers*. ESP ’97. Alexandria, Virginia, USA: ACM, 1997, pp. 180–208 (cit. on p. 11).

Online sources

[15] John Azzolini. *Introduction to Systems Engineering Practices*. 2001. URL: https://www.cesames.net/fichier.php?id=80 (visited on 07/06/2016) (cit. on p. 12).

[16] *Component-based software engineering*. URL: https://en.wikipedia.org/wiki/Component-based_software_engineering (visited on 07/16/2016) (cit. on p. 7).

[17] *Construct 2 Website*. URL: https://www.scirra.com/construct2 (visited on 07/24/2016) (cit. on p. 7).

[18] *Deutsch limit*. URL: https://en.wikipedia.org/wiki/Deutsch_limit (visited on 07/06/2016) (cit. on p. 11).

[19] *Flow Based Programming*. URL: https://github.com/flowbased/flowbased.org/wiki/ (visited on 07/14/2016) (cit. on p. 15).

[20] *General Purpose Simulation System (GPSS)*. URL: https://en.wikipedia.org/wiki/GPSS (visited on 07/03/2016) (cit. on p. 13).

[21] Thomas Ingham. *Using Blueprints in Unreal Engine 4*. URL: http://martiancraft.com/blog/2014/07/blueprints-unreal-engine/ (visited on 07/26/2016) (cit. on p. 19).

[22] *JSON*. URL: https://en.wikipedia.org/wiki/JSON (visited on 06/27/2016) (cit. on p. 17).

[23] John Paul Morrison. *Flow-Based Programming*. URL: http://www.jpaulmorrison.com/fbp (visited on 07/06/2016) (cit. on p. 14).

[24] John Paul Morrison. *NoFlo vs. "Classical" FBP*. URL: http://www.jpaulmorrison.com/fbp/noflo.html (visited on 07/20/2016) (cit. on pp. 16, 45).

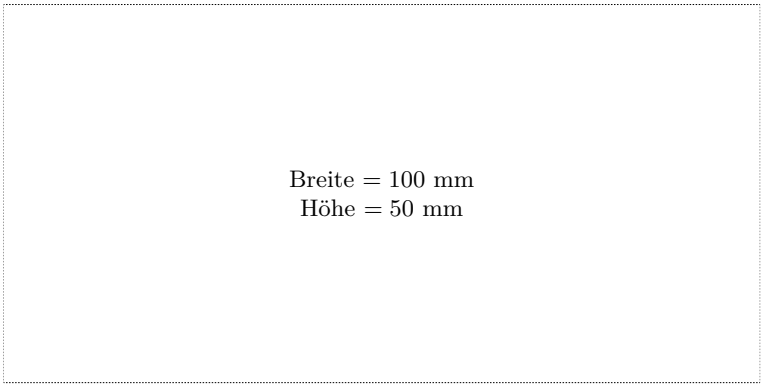
[25] John Paul Morrison. *Software on FBP Website*. URL: http://www.jpaulmorrison.com/fbp/software.html (visited on 07/06/2016) (cit. on p. 13).

[26] *noflo-jekyll*. URL: https://github.com/the-grid/noflo-jekyll (visited on 07/26/2016) (cit. on p. 17).

[27] *Programming paradigm*. URL: https://en.wikipedia.org/wiki/Programming_paradigm (visited on 06/25/2016) (cit. on p. 13).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —