The Reality Editor:
An Open and Universal Tool for Understanding and Controlling the Physical World

by

Valentin Heun

Diplom Designer
Bauhaus Universität Weimar, Germany 2009

Master of Science in Media Arts and Sciences
Massachusetts Institute of Technology 2013

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Media Arts and Sciences at the Massachusetts Institute of Technology
September 2017

2017 Massachusetts Institute of Technology. All right reserved.

Author
Valentin Markus Josef Heun, MIT Program in Media Arts and Sciences

Certified by
Pattie Maes, Professor of Media, Arts and Sciences, Program in Media, Arts and Sciences
Thesis Supervisor

Accepted by
Pattie Maes, Academic Head, Program in Media, Arts and Sciences
The Reality Editor:
An Open and Universal Tool for Understanding and Controlling the Physical World
by
Valentin Heun

Submitted to the Program in Media Arts and Sciences School of Architecture and Planning on August 7, 2017 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Media Arts and Sciences at the Massachusetts Institute of Technology

Abstract

In a future where every physical object has the ability to compute and connect with other physical things, we have to rethink our present user interfaces and interaction metaphors. The desktop metaphor used in personal computers and smartphones was invented for data organization and is not well suited for interaction with things in the physical world. As a result, the growing number of interconnected things (or Internet of Things devices) surrounding us are becoming hard to operate. Each IoT device requires a different app to control it and forces the user to memorize a unique connection and interface. In addition, connected things made by different companies cannot easily be connected to one another. This thesis introduces a novel, directly mapped user interface for connected things built upon World Wide Web technology, a decentralized networking infrastructure for connected things to talk to each other, and a simple, visual user interface for understanding and controlling the connected things around us. The overall system is called the Reality Editor, an open-source, freely and publicly available tool for creating ecosystems of interconnected things. The thesis discusses the design of the Reality Editor, its core ideas and implementation details and a series of real world prototypes that were built to evaluate and improve the tool.

Thesis Supervisor: Pattie Maes
Title: Professor of Media Arts and Sciences, Program in Media Arts and Sciences
The Reality Editor:
An Open and Universal Tool for Understanding and Controlling the Physical World
by
Valentin Heun

The following people served as readers for this thesis:

**Thesis Reader**

Prof. Hiroshi Ishii
Jerome B. Wiesner Professor of Media Arts and Sciences
Program in Media Arts and Sciences

**Thesis Reader**

Mitchel Resnick
LEGO Papert Professor of Learning Research
Program in Media Arts and Sciences
Acknowledgments

Pattie Maes, my advisor, supported this project from a small idea up to the dissertation that it became. It has been a privilege to be in Pattie’s group, learn from her, and think about the future of Human Computer Interaction. I also want to thank Hiroshi Ishii, from whom I had the privilege to learn about the relationship between the physical and the digital, and Mitch Resnick for helping me to explore new pathways and transform my thinking.

I am grateful for the chance to learn from these professors and thankful to be a part of the MIT Media Lab community, without whom the Reality Editor would not have been possible.

In particular, three friends from this community helped bring the Reality Editor to life: Shunichi Kasahara, whom I had the chance to work with very early in the process, when the first ideas for the Reality Editor started to form. Benjamin Reynolds and James Hobin, who joined the project as UROPs, stayed with me all along.

My gratitude also goes to the UROPs who have helped over the years: Kevin Wong, Michelle Suh, Shuo Yan, Andrew Mendez, Eva Stern-Rodriguez, and Afika A. Nyati.

I want to thank all my family and friends for shaping the path of my life and all the teachers and mentors along this path who believed in my abilities.

Finally, my gratitude and thankfulness go to my wife Anissa. Thank you for taking this journey with me.
Table of Contents

Abstract ............................................................................................................................................ 2

1. Introduction ................................................................................................................................... 10
   1.1. Motivation .................................................................................................................................. 10
       1.1.1. User Interface for IoT .............................................................................................................. 10
       1.1.2. Intuitive Method to Materialize Ideas ....................................................................................... 11
   1.2. Approach .................................................................................................................................... 14
       1.2.1. Research ................................................................................................................................. 15
       1.2.2. Design of an Essential Solution ............................................................................................... 15
       1.2.3. Software and Hardware Development ...................................................................................... 15
       1.2.4. Review via Open Source Community and Workshops .................................................................. 16
   1.3. Related Work ............................................................................................................................... 16
   1.4. Roadmap to the Thesis ................................................................................................................ 18
   1.5. Contribution ................................................................................................................................. 19

2. Design Considerations ...................................................................................................................... 20
   2.1. Usability ........................................................................................................................................ 20
   2.2. Stability ....................................................................................................................................... 21
   2.3. Self-Descriptive ........................................................................................................................... 23
   2.4. Robustness ................................................................................................................................... 24
   2.5. Compatibility ............................................................................................................................... 24
   2.6. Ownership .................................................................................................................................... 25
   2.7. Privacy ......................................................................................................................................... 26

3. Core Ideas ......................................................................................................................................... 27
   3.1. Directly mapped digital interface ............................................................................................... 27
   3.2. Bi-Directional Augmented Reality .............................................................................................. 31
   3.3. Unified Communication Protocol ............................................................................................... 33
   3.4. Decentralized Networks ............................................................................................................... 39
   3.5. Web Technology for Augmented Reality Content ....................................................................... 40

4. System Implementation .................................................................................................................. 42
5. Interface Functionality ................................................................. 70

5.1. The Main Interface Buttons ......................................................... 70
  5.1.1. Interacting with a single Object ........................................... 71
  5.1.2. Connecting Objects .......................................................... 71
  5.1.3. Virtual Pocket ..................................................................... 72
  5.1.4. Settings .............................................................................. 72
  5.1.5. Freeze ................................................................................. 73

5.2. Memory Bar and Visual Pointer ................................................. 73
  5.2.1. Memorizing ......................................................................... 74
  5.2.2. Retrieving ........................................................................... 75
  5.2.3. Deleting .............................................................................. 76
  5.2.4. Visual Pointer ..................................................................... 77

5.3. Object Locking ........................................................................... 80

5.4. Developer Tools ......................................................................... 81
  5.4.1. Reality Editor UI ................................................................. 81
  5.4.2. Reality Editor Server ............................................................ 84
  5.4.3. Implementing a New Connected Object .............................. 85

5.5. Touch Connections ................................................................. 86

5.6. Logic Crafting .......................................................................... 88
  5.6.1. Design Considerations .......................................................... 105
  5.6.2. Handling Logic Nodes .......................................................... 113

5.7. Instant User Interface Elements .................................................. 114
5.8. Color Schemes ........................................................................................................... 122

6. Use Case Prototypes ..................................................................................................... 124
  6.1. Lego as Physical Building Block ............................................................................. 124
  6.2. Smart Grid .............................................................................................................. 127
  6.3. Health .................................................................................................................... 128
  6.4. Museum prototype ................................................................................................. 129
  6.5. Lights demonstration ............................................................................................. 130
  6.6. Arduino demonstration .......................................................................................... 131
  6.7. Audi Car demonstration ....................................................................................... 132
  6.8. Virtual Objects ...................................................................................................... 135
  6.9. Lego Mindstorms prototype ................................................................................ 137
  6.10. Lego WeDo prototype ........................................................................................ 138
  6.11. Secure Transactions ............................................................................................ 140
  6.12. Visual Search ....................................................................................................... 141

7. Workshops and Community ......................................................................................... 144
  7.1. First Workshop ....................................................................................................... 144
  7.2. First Release and the Web Forum ......................................................................... 145
  7.3. Second workshop .................................................................................................. 148
  7.4. Third Workshop ..................................................................................................... 149
  7.5. Royal College of Art Workshop .......................................................................... 150
    7.5.1. Bookmark by Fiona O’Leary ............................................................................. 152
    7.5.2. Pill Box by Václav Mlynář .............................................................................. 153
    7.5.3. Card Key by Kyungmin Han ......................................................................... 154
    7.5.4. Augmented Communication by Kristian Knoblauch .................................. 155

8. Discussion ...................................................................................................................... 157
  8.1. Reality Editor: A Critical Discussion from the Perspective of Affordance ............. 157
    8.1.1. Affordances of Connected Objects ............................................................... 158
  8.2. Security and Safety Concerns .............................................................................. 167
  8.3. A Person as a Connected Object .......................................................................... 168
  8.4. Voice Interfaces ..................................................................................................... 169
  8.5. The limitations of the Reality Editor ..................................................................... 170
8.6. Multi-User Scenarios ........................................................................................................... 171
8.7. AI vs GUI .......................................................................................................................... 172
8.8. Standardization ............................................................................................................... 173

9. Conclusion ............................................................................................................................ 175

10. List of Figures ..................................................................................................................... 178

11. Reference ............................................................................................................................. 185

12. Appendix ............................................................................................................................. 192
   12.1. AR User Interface Reference API ................................................................................... 192
   12.2. Hardware Interface Reference API ................................................................................ 194
   12.3. Arduino Reference API ................................................................................................. 197
1. Introduction

1.1. Motivation

Look at the world around us. Physical things seem to be mostly static, and they have little ability to share any data other than the shapes and surfaces that we can see and use. Everything around us relates to a vast amount of data that could be attached. Such data could be a construction plan, a manual for how an item is to be used, and in case of a digital object the code that enables embedded functionalities or even memories that people generate while using these things. Some things are designed to have their functionalities directly combined with other things, while we combine others to tell stories about how we use them. For example, a kitchen knife, a cutting board, and a pan can form quite a story when food is processed. Recently, we can identify two growing technological developments that promise to disrupt the way we see and use the meaning of physical things. These two technologies provide the motivation for this thesis. One is the ability to network physical things into a so-called Internet of Things (IoT). The other is the ability to overlay digital information onto the physical world using augmented reality (AR) enabled by computer vision.

This thesis documents the design, successive implementations and prototyped use cases of the Reality Editor. The Reality Editor is a tool that allows users to perceive the hidden digital information related to the things that surround us, and as a tool that allows us to make meaningful connections among things to form new stories and create new patterns of use. The Reality Editor aims to enable a richer interaction with physical things and more freedom in how we communicate our ideas of use to these things.

1.1.1. User Interface for IoT

As this thesis is being written, the Internet of Things (IoT) is a rapidly growing, complex system of connected sensors, objects and devices, driven by software and services tailored to the goals of corporate customers, their services and products. It is used extensively in coordinating worldwide, decentralized production cycles via distributed sensors and connected robotic
automation. Furthermore, the IoT helps companies collect big data for optimizing business decisions and products. From a non-expert perspective, there is, unfortunately, no unified and intuitive interface for understanding and controlling the world of interconnected things. Typically, a connected thing (or smart device) such as the Nest thermostat [1] or Philips Hue lamp [2] comes with its own unique app for remotely controlling the object from a smartphone or collecting and analyzing data from the object. While this approach may be sufficient for today’s world, where users have only a few connected things in their homes, it does not allow them to take full advantage of the potential of a truly interconnected network of things. As the network scales, using it becomes cumbersome—first, the user must install, learn, and master a separate app for each object, and second, the user must remember which app to open to take control of a particular thing. Memorizing these connections becomes overwhelming as the number of controllable things in our lives grows. Finally, these apps do not allow different connected things from various vendors to talk to one another and therefore limit what a user can accomplish. For those connected things that can speak to one another, there are strict rules in place on how they can communicate.

This thesis presents a future in which the Internet of Things presents a more unified user experience. A future in which the IoT is highly ubiquitous and crosslinked, where all things that are electrically operated in a home, building or an entire city can be connected and interacted with in a simple, uniform way. As the Reality Editor is an augmented reality tool, many of the findings presented in this thesis do not only improve the Internet of Things, but also enable better creation and interaction with other augmented reality disciplines.

1.1.2. Intuitive Method to Materialize Ideas

In order to enable a future in which the IoT is ubiquitous and crosslinked, simple and unified interaction methods need to be introduced that minimize memory and mental load, i.e. they must not be overly abstract and must be self-explanatory. Intuitive methods for materializing logical ideas among physical objects will lower the barrier for non-experts to make meaningful individual connections between the things they own and discover possible new services. The lack of such intuitive methods is seen in modern data and network silos, which are implemented with
proprietary communication protocols and proprietary services, and make it hard for users to discover or create new IoT experiences.

Looking back into human-computer interaction history, a similar situation has occurred before. At the beginning of the personal computing era, before the introduction of graphical operating systems that make use of a desktop metaphor, only experts could master the functionality of computers because of their unwelcoming command line user interfaces. Commands had to be memorized and typed into the terminal for execution when needed. With the advent of the desktop metaphor used in graphical operating systems such as Amiga OS (Figure 1), Mac OS or Windows, suddenly everyone could use a personal computer with little or no training, as the desktop metaphor made it easy to understand how to operate it even without a manual. A user no longer needed to memorize commands, but would find them self-explanatory, thanks to icons and folders on the virtual desktop. For example, a file has the appearance of a piece of paper, which can be deleted by dragging it into the virtual trashcan. Today a computer is in every home and every pocket. The desktop metaphor unlocked the full potential of personal computing. By removing the need for a user to memorize commands, it also reduced the mental load and expertise needed to operate the computer and link to other computers.

*Figure 1 A screenshot of the Amiga OS 1.2. 1986 captured via emulator software*

The desktop metaphor not only allowed non-experts to master the computer, but it provided the basis for software that would enable users to connect with other people and computers. Where
initially experts would control and view private networks, the exchange of data on a public, open and global network enabled all sorts of new uses and services. The development and spread of the Internet protocol TCP/IP [3] allowed formerly incompatible networks to exchange data and as such to form the Internet that we know today. A desktop computing application called the World Wide Web, with its HyperText Markup Language (HTML) [4], the Hypertext Transfer Protocol (HTTP) [5], and the Uniform Resource Identifier (URL) [6], helped jumpstart widespread Internet use. At the time, these technologies contributed to a breakdown of private information and data silos, providing a unified simple and open standard protocol and visual language for data exchange. This infrastructure has been in use ever since. The desktop metaphor—an intuitive interface that can be operated by a non-expert user—is a key underlying factor that enabled the widespread use of the World Wide Web.

This history reminds us of the importance of providing the right user interface for a task. The desktop metaphor was invented with a certain use in mind, namely for the user to master digital documents and software whose function is similar to work on a physical desktop. Today this desktop metaphor still informs the graphical user interfaces of our smartphones. However, the physical world of smart objects does not map naturally to the metaphor of a desktop. Instead, I will argue that it more naturally maps to an interaction with physical and spatial affordances [7]. The icons, windows, apps and menus present in a traditional desktop are missing appropriate components to represent the newly possible digital interactions with IoT objects in the physical world. The goal of this thesis, therefore, is to propose a new metaphor where every physical object has a virtual counterpart that provides the interface to the data and services related to that object and interaction methods that enable a more intuitive and universal digital interaction with connected physical things—an interaction that is scaleable to a truly ubiquitous distribution. These new metaphors and methods must seamlessly merge digital and physical interaction with connected things (or smart devices). They must support the user with intuitive interactions considering spaces and objects within these spaces.

We learned from the history of personal computing how relevant meaningful and simple interaction metaphors and user interfaces are. Given the advent of the ubiquitous Internet of Things in our daily lives and workspaces, it is the goal of this thesis to rethink present IoT
interfaces and technology in a similar fashion. The central question of this thesis is: What kind of user interface and what kind of supporting infrastructure will open up a similar potential for IoT as the desktop metaphor implemented with graphical operating systems and Internet protocols did for personal computing and the World Wide Web? What kind of protocols and standards can break down the data silos that control the IoT today? And what type of interface can become the equivalent of the desktop metaphor and browser for the connected world that we increasingly live in? The vision that motivates this thesis is clear: an interface that allows us humans to materialize logical ideas in the physical space around us, merging the digital and the physical into one reality that appears to us as inseparable from one another. This new definition of reality will help us engage more easily with the physical world around us and will prevent us from being bound to desktop-like machines.

1.2. Approach

This thesis was developed through an iterative process that consisted of a repetition of four steps (Figure 2).

![Figure 2](This diagram shows the four different steps of the approach taken for this thesis)
1.2.1. Research
The thesis started with research into the problem and a definition of a possible solution. Some example questions that were addressed include:

1. What defines the entire problem?
2. How can it be broken down into smaller problem components?
3. What prior inventions have been proposed to solve the problem or similar problems?

1.2.2. Design of an Essential Solution
There may be many approaches that constitute a satisfactory solution. However, the solution not only has to work, but it also must solve the problem elegantly and reach the essential core of the problem. If there is a slightly better solution available, it must be explored. Eventually the goal is to reach a solution that is easy for a user to understand and operate. In my eyes, a Human Computer Interaction (HCI) problem is only solved if a user can easily understand the problem as well as the solution. In such a case, the entry barrier for a user will be low and the learning curve will constantly allow for simple learning.

1.2.3. Software and Hardware Development
Once an essential solution has been designed, it must be realized as a useable software and hardware demonstration. Since this thesis describes human-computer interactions, and HCI is strongly dependent on the user experience and user understanding, only a fully functional demonstration of the system can provide proof of its usability. Software and hardware must tell the full story of the interaction, to test the solution and find new ideas that may improve it. Therefore, a major part of this thesis is the development of a fully functional, decentralized operating system for connected objects, and a user interface that allows the user to curate the connections among objects, similar to the way a conductor would conduct an orchestra. Connected objects are physical things that have embedded technology that allows them to connect via a network and communicate with the Reality Editor. Connected objects also provide a visual marker that allows the Reality Editor to superimpose user interfaces onto them.
1.2.4. Review via Open Source Community and Workshops

Once a solution is implemented that provides the desired user experience, it is shared with the open source community for evaluation and testing. For this purpose, the software is organized in a modular fashion, so that individuals in the open source community can easily lead development and improvements of different parts of the software project. This development model allowed many of the system errors, bugs, and flawed designs to be discussed in a peer-reviewed community. Additionally, peer review workshops were held to test how designers and developers could use the Reality Editor to develop their projects. The open source community peer review process and the workshops resulted in significant findings that led to new problems and therefore new research. And thus, the cycle was repeated.

1.3. Related Work

The Reality Editor is a platform that touches a multitude of categories. The three most important categories are augmented reality, Internet of things and networking. This chapter outlines essential related work in these categories. Other related work will be referenced throughout the thesis.

The area of augmented reality has a wide span, from the first experiments in AR and virtual reality by Rosenberg [8] and Sutherland [9] up to present commercial successes in mainstream advertising [10] and apps for virtually placing furniture in one’s home as a part of the shopping experience [11]. The latter represents a typical use case for which present authoring platforms are built [12]. Advertising, maintenance and contextual guidance for assembly tasks [13] have been a strong focus for authoring platforms over the years and throughout the AR community [14]. Most of the commercially available AR systems are built using authoring tools such as Unity 3D [12] and Layar [15]. While these platforms enable easy content creation, at this point there is no AR authoring platform available for the public that supports the creation of AR interfaces for the Internet of Things (IoT) and that is also based on open communication standards. This means that their content creation, content delivery and content consumption methods are proprietary and cannot be modified or implemented by others. Furthermore, state-
of-the-art platforms provide limited support for merging the physical with the digital, as they mostly focus on superimposing content onto static markers or objects.

There is for example a mobile augmented reality authoring platform for annotating physical objects introduced by Kasahara et al. [16]. Pucihar et al. [17] introduced an authoring platform that allows annotation via a “contact-view” gesture. However, in general augmented reality (AR) authoring platforms use the desktop computer as a content generation tool. There are some authoring platforms that use HTML internally to render content [18][19][20]. There are commercially available tools [15] as well as more specialized platforms, such as those supporting furniture assembly [21]. Others have used more proprietary technologies [22][23] or have announced proprietary platforms [24] which eventually led or will lead to incompatibilities.

Outside of the scope of authoring platforms, researchers have created tools to control the immediate environment via AR as a hybrid user interface [25] with automatic network infrastructure [26], collaboration frameworks [27], or as a control system for programming everyday tasks performed by a household robot [28]. Such interactions have been further explored by Kasahara et al. in ExTouch [29], an AR interface offering a vocabulary for moving objects. Until now, mixed reality and merged reality explorations [30][31][32] required complex technical development and a tight coupling of the graphical interface and the physical objects. Such research projects will benefit from the platform described in this thesis.

As a foundation for the presented AR platform, the vision of the Internet of Things [33] shows how the interconnectivity of the World Wide Web can be of use in our material world, by adding digital control over an increasing number of physical objects [34]. We have seen the introduction of systems that enable simple communication with Internet-connected physical actuators and sensors [35], a physical equivalent of widgets [36], and consumer products that enable home connectivity and automation [37]. Researchers have created IoT toolkits [38] that enable simple creation of connected experiences, as well as data flow programming interfaces for IoT objects [39][15]. However, none of these IoT interfaces offers an intuitive augmented user experience, since the well-known desktop paradigm is not directly transferable to physical space [40]. As a
result, we see technology that tries to predict user demands instead of offering users better control [1], thereby avoiding one problem by introducing another.

We also need to consider existing networking methods. Some network technologies try to simplify the connectivity of everyday objects [41] or even reinvent the network infrastructure for the specific demands of IoT [42]. Other technologies implement distributed computer agents [43] that can execute distributed software. Since the software manages the communicated data among the nodes, the system is highly dependent on homogeneous runtime environments and methods for defining distributed states. This thesis describes alternative methods for distributed software; because they act on a unified networked data model, they do not require homogeneous runtime environments and take care of distributed states.

From a network infrastructure perspective, the Internet already has well-functioning technologies that can easily be adapted for IoT products [2]. From an IoT perspective, hardware and network systems have been explored in depth. Unfortunately, user interfaces that effectively and easily connect with their physical counterparts, tools to create such interfaces, and simple networking infrastructures to support such interfaces are missing. This thesis introduces the Reality Editor, consisting of such a platform, tools and networking infrastructure.

1.4. Roadmap to the Thesis

This thesis is organized into nine chapters. After this introduction, a second design consideration chapter follows, which provides a philosophical outline of the core ideas described in chapter three. The core ideas are complemented with a system implementation in chapter four. The interface functionality chapter explains user interfaces that built upon the core ideas and the implemented system. The sixth chapter introduces key demonstrations that were built while developing the entire system and all interfaces. The Reality Editor was released to a public open source community, and multiple workshops were held. The seventh chapter describes findings that emerged from the open source community forum and the workshops. A discussion chapter discusses remaining open questions that were not answered with software implementations, demonstrations, and workshops. The last chapter provides a final conclusion for this thesis.
1.5. Contribution

The contribution of this thesis is a system that makes the physical world tinkerable and therefore invites users to readjust and reinvent how physical connected things operate. This tinkering ability is implemented with a system that is strongly influenced by the philosophy of the World Wide Web. A clear definition of connected objects describes objects that have augmented reality user interfaces and the ability to network. Built upon this definition, the thesis contributes a distributed platform for connected objects that implements a unified communication protocol for decentralized networking and directly mapped digital interfaces that allow the programming of connected objects with bi-directional augmented reality. The entire platform is implemented with web technology and a novel method is presented that allows ordinary webpages to be used as augmented reality interfaces for connected objects. The entire source code of the presented platform is open source and publicly available.

Furthermore, this thesis contributes improved user interfaces and tools that allow developers and users to create and interact with connected objects. A novel programming tool allows the user to program the behavior of connected objects and the relationships between them.
2. Design Considerations

These are general guidelines that we believe need to be considered when building a system that facilitates interaction with physical objects that have a digital counterpart. When digital, and often invisible, content influences the affordances and behavior of physical things, the shared common cultural memories of how to interact with the world are disrupted. Users expect certain properties based on their familiarity with regular physical objects, which are different from their expectations for software. These include aspects of usability, long-lasting functionality, ownership, security, coherent design language, simple interaction paradigms, and privacy.

This chapter describes these design considerations. They motivated the choices made in the design and implementation of the Reality Editor and generally provide a guideline for discussion throughout this thesis document. There are other questions that come up once the Reality Editor is in use, such as the interaction with objects in a social context. However, as these questions emerge out of these design considerations and the resulting platform implementation, they have been moved to the discussion at the end of the thesis.

2.1. Usability

Naturally, simple usability is key to the user when interacting with physical objects. Any object that operates in physical space needs to follow this expectation [44]. A system that extends such interactions into the domain of the digital must eliminate all needless steps for interaction and setup and provide a similarly easily legible set of functionalities. A connected object must feel like a normal physical object. In the best-case scenario, a user would just buy, unpack, and immediately start to interact with such a connected object. Users should be able to immediately make usage of the digital functions and define how this newly acquired object relates to the other connected objects they already own. Therefore, such a connected object should be auto-discoverable and fit right into the existing ecosystem of connected objects.
In a best-case scenario, the object will not depend on a single ecosystem that does not relate to other ecosystems. This can be achieved with universal open standards that are simple to implement and an architecture that allows to translate among different communication methods. The complex processes of discovery, network communication and data handling must work automatically in the background, so that the system appears simple to the user and the developer alike. This automation should never compromise the level of user control over the functionalities that are user-facing or the functionality that defines the user data handling. Instead, the system and the connected object only interact to mediate a preparation for the final action taken by the user. If automation is implemented to such a degree that the object is acting on behalf of the user, the user must fully understand and accept it in each individual case.

As a GUI, the Reality Editor must maximize the user’s understanding of cause and effect. This is possible with an interface metaphor that combines the physical functionality with a virtual user interface as well connecting methods that imitate the real-world behavior of using wires to connect physical objects. For example, a user can hold the Reality Editor onto a light and instantly see a virtual interface attached to the light. A user can then drag and drop with a Reality Editor a line from a light switch to a light bulb. As will be explained later in this thesis, this action clearly specifies that the switch will control the light. However, the light and the switch mediate a connection in the background that is kept alive at all times, until the user performs actions that would change the connection.

2.2. Stability

When comparing physical objects with software, the financial value of a physical object can easily exceed that of digital software. Physical objects typically have a much longer lifetime and consume more resources in their production compared to software. If the ability of objects to connect with each other becomes truly ubiquitous, then we will eventually have augmented artifacts passed on through generations within a family. This is a fundamentally different behavior compared to how we treat software or even web services, where the software only works for as long as the operational system still exists. For example, a future child could find a
toy in his grandparents’ attic that includes the functionality to be connected. If the software foundation for this connected object is well-designed, so that it is resilient to platform and system changes, the child might be able to just power up the inherited toy and include it in his own environment. Like photos, physical objects hold memories [45] and therefore require a long lifespan [46]. These physical objects should be easy to adapt, hack, or upcycle, as this guarantees that they can be reanimated with little effort. Comparing such objects with the lifespan of other memorabilia, ideally connected objects should be developed with a lifespan of 50 years or more in mind.

To serve such a lifespan, the design of connected objects must factor in two important aspects. For one, they must be independent from external systems and external infrastructure. Second, they must make use of the most common and simplest technologies used over a long period of time. In the development of this thesis, many different computer languages and authoring tools have been used. For example, an early version was developed with OpenFrameworks [47] based on C++ with a central server structure. For a later version, Unity was considered. However, there were strong reasons to reject development with proprietary software. Earlier augmented reality authoring tools were built with proprietary closed software tools, and once these software tools disappeared from the market, the authoring tools disappeared as well [23]. The World Wide Web is an exception in that it provides all of the beneficial properties that serve the described needs—consistency, compatibility and open standards. HTML[4] and CSS are likely the best-known graphical user interface standards, and the same goes for the networking protocols powering the Web, such as HTTP [5], WebSockets [48] and TCP/IP [3]. These enduring foundations are an inspiration for the Reality Editor and its services. It is still possible to browse the very first webpage that Tim Berners-Lee created in 1990 [49]. Similarly, we should still be able to view and interact with augmented objects in 50 years. If connected objects are built on the same foundation as the web, we are likely to be able to do the same with upcoming slew of mass-produced connected objects.
2.3. Self-Descriptive

Connected objects for the home user need to be freed from any centralized external services. This means that all of the information about connections to other connected objects, all user interfaces, and the entire logic needs to be stored within the objects themselves. External services are often bound to companies, market rules, and financial gains. This can lead to discontinuation of the services that allow connected objects to work. Only if objects are independent from external services are they guaranteed to function in 50 years. Therefore, each object must carry every piece of information it needs, stored in its own memory, including the visual references used for tracking its external appearance.

Not only must all of this data be stored in the object itself, it also must be stored in a common and open format that will survive into the future. In the case of visual references, JPEG and PNG are common and open formats. Code and hypertext is stored in Unicode-encoded text files. Code must also include its source code, which is the case using JavaScript. Proprietary formats can also be stored in the connected object, but must be accompanied by a file that describes the same content in an open format. For example, the tracking framework used in this thesis uses a closed proprietary format for storing information about natural features; however, a JPEG image with the original source image for the features is included as well. This approach guarantees that anyone will be able to build software to interact with the object, therefore increasing the chance that long after the Reality Editor has lost its user base, the objects will remain functional or can be reanimated.

The importance of this design principle is demonstrated in the problems with the DART [23] platform, which was dependent on the now-closed authoring system Director from Macromedia. Director was discontinued and all applications built for DART became useless. With the Reality Editor this will not be possible, since new systems can be built to interact with the Reality Editor and its services, thanks to the open standards, API and code.
2.4. Robustness

Physical, non-electronic objects work as long as they are structurally and mechanically intact. They are mostly independent from centralized systems such as electricity or communication networks. As such, non-electronic objects are protected from external influences which makes them very robust. For example, a bicycle will continue to work perfectly even without electricity, an Internet connection, or the presence of other bicycles. Augmented objects should behave in the same self-sufficient way, if they are to avoid malfunctioning and being abandoned. If a connected heating system stops working because of an Internet connectivity outage during a blizzard, the result could be fatal.

To protect such systems from malfunction and guarantee maximum functionality, connected objects must communicate directly between sender and receiver with a minimum number of intermediate routes. Introducing third-party services to the communication flow reduces stability, introduces security risks, and compromises functionality. A light switch that relies on an Internet service provider and a cloud service to switch on a light will stop working when any of these services change their protocols or go out of business. Additionally, if the light switch and light are permanently exposed to the Internet, they require permanent protection from hacking attacks. However, if the light and the light switch only function in a local network and only accept network connections from a local network, the security aspects of such a system change for the better. The Reality Editor must provide decentralized services that work without a link into the worldwide Internet. As such, these services can be considered decentralized networks in which objects always communicate directly with each other.

2.5. Compatibility

One problem that occurs with the plurality of connected things is defining compatibility. Usually a consortium of manufacturers defines all of the options that need to be understood by all of the devices on the network, so that they can talk to each other. One difficulty in this process of development is making sure that a connected object produced today will work with a connected object that is generated in the future. Since connected objects can last for years, and we hope that
they will last even longer, one would need to know all possible attributes of all future objects to ensure continued compatibility.

Connected objects must connect in a way that avoids exponentially growing compatibility issues. The use of augmented reality can provide a novel solution. Because augmented reality allows users to visually understand all the components (inputs and outputs) of a connected object, they can establish connections between subcomponents of different objects. These subcomponents, however, can be modeled with a generic data format that remains the same at all times. As such, the only component that changes in the connection is the user interface of a connected object itself. Since this interface is stored in the object, the data communication among objects stays generic and universal. As a result, augmented reality allows the creation of connected objects that share a simple communication protocol for potentially long-lasting compatibility.

2.6. Ownership

Physical objects differ from everyday consumer software or Internet services, as they may have a high financial acquisition value. Some high-value items will be invested in only once or twice in a human lifetime, such as houses, heating systems, or even cars. Usually, the owners of physical object have the right to do whatever they desire with them [REF]. When it comes to objects that contain software as well, however, this ownership is suddenly in question. In most cases, a user cannot claim ownership of proprietary software beyond acquiring a license to use it. Since a connected object needs to work in a stable fashion at all times, and users must feel ownership for the physical shape of connected objects, such objects must not be bound to software license agreements that generate shared ownerships. Instead, software license agreements for connected objects must be entirely free or open source and granted for the lifetime of the object. To illustrate this need, we can think of a heating system. If the software that drives it is dependent on an external service, the external service may suddenly charge higher fees or even discontinue its business, causing severe damage. Such changes, or even more harmful actions, are well within the rights of software providers. From the perspective of the owner of a physical object, however, particularly one that represents a substantial financial investment or a vital system,
these rules are unimaginable. Therefore, the Reality Editor services must be entirely open source and free for anyone and for any purpose.

2.7. Privacy

Physical things operate in a private zone, unlike Web services that are accessible by the public. In contrast to software, personal physical things cannot be copied, instantly relocated or touched by anyone in the world. Rather, a single instance of an object is usually secured in one’s home or workplace, or on one’s body. Within that space, things are freely accessible. The key for the front door becomes the security feature for everything behind the door. The same principles should apply to connected objects as well. Guarded within a private personal network, the objects should be able to communicate freely with each other. Every user who has been granted access to this private network should be considered a trusted person with access to the connected objects in that home or property. The router that connects the private network with the Internet becomes the property’s door. It should provide a maximum of protection. However, a locking mechanism should allow the owner of the connected things to define whether other members of the network are allowed to tamper with the objects’ functionality. This can be thought of like storing objects in a chest with locked drawers. A user may connect properties of a connected object to an Internet service, but in this instance, the service must be sandboxed and its access limited to a single data point.

All of these design ideas have to be considered when building connected objects that are not software and not a physical object alone. How is it possible to build a platform for connected objects that implements these considerations? The next chapter describes a group of core ideas that help to implement what was described in this chapter.
3. Core Ideas

This thesis combines a set of core ideas into a tool called the Reality Editor that creates a simple and unified interface for the Internet of Things while respecting the design considerations described in the previous chapter. These core ideas are: (1) A direct mapping of digital interfaces onto physical objects; (2) bi-directional augmented reality for synchronizing the digital and physical representation of an object; (3) a simple unified data communication protocol; (4) a decentralized networking infrastructure and auto-discovery methods for objects to find and talk with one another and (5) ordinary web content that is 3D-transformed using augmented reality.

3.1. Directly mapped digital interface

In order to solve the central problem of today’s user interfaces for the Internet of Things, namely the necessary memorization of connections between digital interfaces and physical objects, one needs to understand what exactly the problem is. Current user interfaces for connected objects
mainly consist of applications that are accessible via icons in one’s smartphone (Figure 3). Each app implements an individual user interface. Yet all such user interfaces consist of lists of items, interface cards, more icons, and interactive elements such as buttons (Figure 4). These user interfaces require that the user builds an abstract model about the connections between the app’s buttons and icons and the objects it controls. For example, one can name a connected object in a list (e.g. “desk light”) or give it an icon (e.g. a light bulb) that matches its physical form. This works well when the connected object is valuable to the user and its operation can be expected to command a certain amount of care and attention. However, it becomes more difficult when scaling this kind of user interface to hundreds of connected objects. Suddenly what was clear becomes opaque, hidden in endless lists of connected objects or a forest of matching icons. Which lightbulb icon belongs to the kitchen light? What was light number 4 in the living room again? The mapping of the physical and the digital becomes overly complex in such a scenario, and even simpler scenarios would likely overwhelm someone who is not already comfortable with such technology.

Figure 4 An illustration that shows dropdown menus in a smartphone on the right side and physical interfaces on the left side.

On the other hand, we can draw inspiration from the world of physical objects for how such mapping should appear. In a room with many light switches, finding the right light switch for a light can take a couple of tries. Since the user interface is disconnected from the actual light, it is
impossible for a user to just know. However, when we look at desk lights or kitchen appliances we see that user interfaces are mapped to where the resulting action happens. As such we can easily switch hundreds of lights on and off without a single failure. This direct mapping makes user interaction intuitive, and it can be adopted in digital interactions with connected objects.

The Reality Editor introduces such direct mapping with a user interface that seamlessly combines the physical appearance of an object with a digital representation of the same object. A digital user interface is projected in 3D space onto a physical connected object using the camera feed of a smartphone and augmented reality technology. Direct mapping allows the user to interact with digital information placed at the spot where the connected object is located. As a result, the user no longer needs to memorize the connection between apps, icons, a list, or menus and physical connected objects. Direct mapping allows users to not only understand better how a physical thing and its digital user interface correspond, but also to operate an unlimited number of connected objects without prior education.

For example, a user can easily operate directly mapped user interfaces for two desk lights (Figure 5) by pointing a smartphone at the lights. Once the lights are in sight, corresponding digital interfaces pop up, visually mapped directly onto the physical lights. Using this approach, digital user interfaces can extend physical user interfaces in their functionality. A physical desk light user interface might only provide an on/off switch, but with a directly mapped user interface, the user can also make changes to the color of the light. Since the augmented user interface for the light is directly mapped onto each object individually, a user never needs to memorize this connection. Hence, a high-rise building with thousands of lights and doors could be operated without the user ever needing to memorize a single name or icon.
In another example, a user can map a person’s vital signs onto the person’s body. Such interfaces can lower the amount of time spent per patient in a hospital. A doctor only needs to point a smartphone at a patient or look at the person through a head-mounted display to see all necessary data. Such direct mapping of data onto patients may reduce errors in data handling.

Another example shows a general-purpose capacitive sensor Figure 6. This sensor uses a direct-mapped user interface to project the sensor readings right onto the sensor. Scaled to hundreds of instances, an inspector could walk through a factory and instantly check all the sensors for accurate readings.
3.2. Bi-Directional Augmented Reality

Looking at the state of the art in augmented reality [50], most applications use interfaces to visualize content and overlay it onto the physical world. As such, this overlay is only a unidirectional augmented reality experience. These interfaces can be directly mapped onto objects and devices in the physical world, but they have no direct influence on the physical world. However, even when looking at the simple, two-light example in Figure 5, something appears different. Instead of only receiving the states of the lights visualized on the physical lights, a user can manipulate the augmented reality user interface to actively change the state of the lights, in this case changing the hue of the lamp. Once the augmented reality experience has ended, the alteration of the physical object remains. The light retains its new color until the user changes it again. Unlike prior augmented reality applications, the Reality Editor establishes a bi-directional augmented reality interaction. The direct mapping allows this bi-directional interaction to be seamless and intuitive for the user (Figure 7).

*Figure 6* An augmented reality application within an iPad. The application shows a capacitive touch sensor and its data augmented onto the sensor.
With such a bi-directional implementation of augmented reality, every action taken in a digital user interface changes the functionality and behavior of the physical object, and every change in the physical object influences its appearance in the digital interface.

Let’s consider another example to explore this bi-directional user interface further. A radio (Figure 8) can have a simple physical user interface that controls volume and the tuning of a selection of songs or playlists. The digital user interface provides extended functionality for the
radio with settings, playlists and songs for the user to choose from. All changes to the physical radio are mirrored in the augmented overlay and vice versa, using the bi-directional augmented reality user interface. This has significant implications for the design of the physical radio and its interface. Instead of having to provide user interface elements for all functionality in a physical form, some may be moved to the virtual interface, thereby simplifying the object. The remaining physical interface elements will be more prominent in the less complex design and therefore easier to understand.

Given a bi-directional coupling of AR and the physical object, a user can even change the functionality of the physical interface elements of the object. In the case of the radio, the user can redefine what the tuning dial does by dragging and dropping a song or playlist onto a specific spot on the dial. When the AR interface is removed, the new settings stay in the radio, and the user can operate it physically. With such use of augmented reality, a user can program and operate an unlimited number of physical things, without increasing the cognitive load required for browsing the user interfaces on a smartphone, as every operation has its spatial spot or is programmed into the physical world.

3.3. Unified Communication Protocol

A bi-directional implementation of augmented reality with directly mapped user interfaces can change our conception of what a physical object is and extend the possibilities for what an object can deliver. The radio example showed how bi-directional augmented reality can affect the physical functionalities and interaction modalities of an object. Looking at the light example, it is evident that one can interact with a potentially infinite number of objects, yet the interaction stays intuitive because a user never needs to memorize associations between the digital interface and physical objects. As such the digital and physical nature of an object are seamlessly interwoven, supporting all of the possible interactions in each domain. An important advantage of the digital domain is the ease of copying and creating instances of an object. Digital objects can be combined to form new objects or reconfigured to serve a different purpose. However, such flexibility is usually difficult to reproduce with a physical object.
In traditional product design, for example, an object such as a microwave may be defined as a unit that comprises a set of buttons, displays and operational units. Altogether, they form the product. This product requires precise planning and testing, because once the blueprints are set and production starts, that product is frozen in its form and meaning. A customer buys this product as it is, with pre-defined functionality that will never change. However, once the digital and physical components of an object are seamlessly combined, these components and their connections can be reconfigured as simply as software, yet they provide physical user interfaces. This results in a high degree of modularity that gives the user options to form personalized solutions for the interfaces and behavior of physical objects. For example, in the case of a Reality Editor-controlled microwave, a user can choose to connect a start button and display not only to the heating element, but also to a multitude of sensors and actuators. For example, the living room lights could indicate when the food is ready, a third-party temperature sensor could control the temperature of the food, and a connected smoke detector could turn off the oven automatically in case of an emergency. A combination of multiple sensors could indicate if the user is away from home or asleep and automatically switch off the oven before any harm occurs.

With a directly mapped user interface, these scenarios can be visually represented with simple lines that connect from an origin property of one object to a destination property of another object. For example, the timing signal property for an oven timer can be represented with a node that allows the user to draw a line from the timer to a node representing the brightness of the living room light. Or a general-purpose switch can connect to the brightness of a desk light to enable operation at a distance (Figure 9).
Figure 9 A screenshot from the Reality Editor running on an iPhone. A push button and a light are connected with a virtual patch wire.

To implement this extended perspective on the functionality of objects, fundamental networking and user interaction problems need to be solved to avoid incompatible data silos and closed ecosystems. A major reason why data silos and closed ecosystems exist today is that current IoT interfaces confront users with a high level of abstraction. This abstraction, and the users’ resulting inability to make self-informed choices, means that manufacturers need to predefine how connected objects can be combined and operated. Without an interface that allows users to understand the connections between apps and objects, they can only choose from certain predefined patterns and behaviors. The limited scope of these predefined settings generates silos and incompatibility with other vendors.

One answer to this problem is to make connected objects into physical avatars of remote services. This means that the object has no embedded functionality but sends all inputs to a remote server. The remote server then responds to the physical avatar with an accurate output. Such full control over the configuration of a connected object allows the manufacturer to update and service the connected object at any time. This allows the vendor to build a network of
contracts with other vendors and establish a connected world. This seems like an easy answer; however, it comes with many disadvantages. For one, this solution results in a power structure in which one provider has control over the infrastructure of many users, instead of the user owning this kind of control. In such an infrastructure, the failure or business change of a single entity (the central service) can result in loss and damage to many households. Another disadvantage is that such a centralized owned infrastructure prevents the development and exploration of new objects, combinations of objects, and services that potentially build upon a network of connected objects, because every change and every added component needs to be approved by a single service owner instead of gaining traction through user interest.

![Illustration for the separation of an object into all of its components, represented as simple floating-point numbers.](image)

The direct mapping implemented in the Reality Editor and the previously described method for connecting nodes provides an interesting alternative solution to this problem. It allows a manufacturer to visually break an object down into its interactive components (Figure 10). Each of these attributes or components can be represented with a single data node. This visual breakdown allows the user to understand the exact functionality of the object and select a
specific data node for the functionality of choice. The user can then connect this data node to a destination data node of another object.

For example, a toaster can be broken down into the components of a heating unit, a setting knob, a start slider and a timing rotation knob. Using the Reality Editor, the functionality of the start slider and timer of the toaster can easily be repurposed to drive the motor of a food processor, because its components include a setting knob and a motor. A virtual patch wire, resembling a physical wire that might be used to connect an amplifier with an electric guitar, can be used in the AR interface to visualize the connection (Figure 9). If the connection becomes obsolete, a user can simply swipe through the wire to delete it (Figure 11).

Figure 11 A screenshot from the Reality Editor running on an iPhone. A push button and a light are connected with a virtual patch wire. A user swipes with a finger trough the patch wire to disconnect the objects.

This visual user interface for connecting objects offers an equally simple communication protocol counterpart. Since each data node represents a single element of the user interface, each data point can be represented with a single number value. A slider can be represented with a floating-point value ranging between zero and one, while a push button can be represented exactly as zero or one. Every sensor can be represented by this method. Every user interface of
higher complexity can be assembled from either sliders or buttons. A touchpad, for example, can be represented as two sliders for the x and y axes, as well as a push button for finger taps. On the other hand, every actuator can be either on/off or represent a range of actuation. The implemented communication protocol sends a single floating-point number with a value in the range from zero to one from an origin to a destination address, thereby connecting two data nodes. The number of decimal places defines the granularity of the signal, whereby the minimum and maximum always represent zero and one. The data format of the protocol is implemented with JSON and the communication layer is implemented with WebSockets. Packaged together, the protocol implementing the connection of data nodes and the transmission of simple universal number values does not require the implementation of new technology in modern Internet infrastructure. Unlike other protocols used for connecting objects in the Internet of Things (IoT), the origin and destination objects do not need any further description to exchange data. This means that the objects describe themselves to a user and the user selects the right connection from among the data nodes. This is fundamentally different from a model where the objects explain themselves to a system that mediates the choices from which a user can select. As such, the visual implementation supporting this protocol allows the user full control over the connections between components.

This control also means that a user can make an informed choice to connect a specific data node to a web service. For example, a user can connect the timing knob of a toaster to a web service and receive a notification once the timer switches the floating-point number from one to zero. Such services can also be used to get time-based information about usage of objects, or the data can be used to suggest a better configuration for an object. Due to the simple protocol and universal nature of the networked data (simple numbers), these web services no longer need to be proprietary to the device manufacturer, but can be open instead. As a result, this generalization can enable true interconnectivity between things and a multitude of services. As such, this infrastructure is similar in spirit to the fundamental Internet protocols, where every user can build a server that implements a service accessible by everyone else. This means, for example, that a manufacturer no longer needs to provide services for its own product, but a multitude of startups and established companies can provide services on demand. Users can make informed choices about what services have access to their own connected objects.
3.4. Decentralized Networks

The design considerations stipulated that a system supporting the Reality Editor must function in the best interests of the user. It must also be private and entirely owned by the user. A special infrastructure is required to serve this purpose. Two assumptions are made for the Reality Editor server. First, objects that are editable with the Reality Editor must be equipped with a strong enough processor and an operating system that can run the server. Second, they must be able to network with other objects and the Reality Editor via TCP/IP and UDP.

These objects are therefore able to build networks amongst themselves in a private home or office where a router handles the network communication. UDP broadcasting can be used to send messages that all participants can receive. As such, each facility (home or workplace) builds a decentralized network of connected objects, which are networking among themselves in a decentralized fashion. However, since the objects “speak” TCP/IP they are also able to communicate with services hosted on the Internet.

The data packages sent with the Reality Editor protocol implement a universal, randomized, and time-sensitive identifier that makes them clearly identifiable without the need for a central registry. Yet this ID is not sufficient for worldwide identification. Unlike Internet of Things infrastructure, the Reality Editor is implemented with private users and their personal needs in mind. Devices only need to connect with each other and the Reality Editor, while logged into a private wireless network. Each object discovers relevant objects and their IP addresses and then matches these IP addresses with the unified identifiers. Once this match is made, the communication can be performed locally or on a world-wide scale via web services. However, the services separate the global and the personal scale very distinctly.

In a local network, the IP/UUID matching is performed with a heartbeat message sent out by every object via UDP broadcasting. This message then allows other objects and the Reality Editor to communicate with each other. With such an implementation, the protocol becomes independent from the network layer and infrastructure, where often identifiers are only granted temporarily. With such an infrastructure, the Reality Editor implements a system that allows the
objects owned by a user to communicate directly, without the need for external networking services on the base level. The system and service implementation will be explained in more detail in the system implementation chapter.

3.5. Web Technology for Augmented Reality Content

Reflecting back on the design considerations, using ordinary HTML pages is an important contribution. Unlike other content platforms such as Unity 3D [12], the Reality Editor does not introduce a new set of specialized knowledge, but extends established solutions. The reuse of HTML for augmented reality enables a broad base of web developers to create content for the Reality Editor. As the Developer Tools chapter will explain, the Reality Editor platform does not require any knowledge of augmented reality technologies. The Reality Editor has built-in tools to support the positioning of HTML content in space. Using the World Wide Web as its rendering foundation guarantees that the Reality Editor will be robust and its user interfaces will survive many technological iterations.

The Reality Editor implements each individual directly mapped user interfaces using these open World Wide Web and Internet standards. WebGL is used to render 3D scenes into the Reality Editor, while simple HTML, CSS and JavaScript are used to render ordinary 2D web content in 3D space to form all of the directly mapped user interfaces. The Reality Editor implements such 3D-rendered 2D web content without modifications to the original source code, which guarantees that all graphical user interfaces are constructed the same way that webpages are created for a standard web browser. Unlike other solutions [20], this implementation enables the Reality Editor user interface to be rendered with ordinary 2D HTML content and the Reality Editor allows to display and 3D-transform a multitude of such 2D HTML content simultaneously. This is possible because the Reality Editor implements iFrames generated just-in-time via HTML to load the content. These iFrames are then transformed with a 4X4 Matrix using the CSS3D-transformation feature that is hardware accelerated in WebKit, the browser technology in iOS. This technology allows objects to map their UI content in 3D space onto an AR marker.
The importance of this becomes evident when looking at the very first webpage ever created. Not only does it look the same as Tim Berners-Lee originally intended, but it can be entirely rendered with no changes in 3D-augmented space within the Reality Editor (Figure 12). The longevity of the World Wide Web is possible because it builds upon open and well-defined standards with its protocols, its pointers (URL), its markup language (HTML), and its programming language (JavaScript). These standards are defined with a maximum of simplicity and general use in mind, allowing the incremental addition of new content with clearly defined APIs that everyone can rely on. Such consistency is a remarkable achievement, from today's perspective, when new technologies disrupt the status quo on a regular basis and significant amounts of “electronic trash” are produced as systems become incompatible. When building digital interfaces and network infrastructure for physical objects, such clearly defined standards will always be relevant. The Reality Editor is entirely inspired by and implements the World Wide Web ideas and architecture. This means that future work will be able to build up on this thesis the same way the first webpage could be rendered as an AR interface with the Reality Editor.
4. System Implementation

The Reality Editor consists of two main components, the Reality Editor mobile application and the server infrastructure. Over time multiple versions have been developed and every version implemented new findings. The first versions for the Reality Editor and the server were implemented with openFrameworks [47]. In this first implementation, the server was centralized to a single server. All logic was taken care of by the central server and the server communicated to the individual connected objects via a simplified protocol. This system was easy to initiate some first experiments with, but over time its structure became more and more complex. During the development of the design foundations it was clear that this system would not uphold the expectations that are required to serve the needs of connected distributed objects. Several follow-up experiments were necessary.

First, I had to discover the right foundation for the Reality Editor mobile application. Initially there was an idea to use the Unity [12] development environment for the Reality Editor. However, Unity seemed to be particularly difficult to use for a couple of reasons. For one, it is a proprietary system. This means that every connected object would become dependent on the direction and success of the Unity system. Another problem was that Unity is not optimized for just-in-time loading of distributed content. Unlike a web browser that is optimized to load and interact with content from many unknown sources, the Unity system is optimized for single-user, single-player experiences. All of the content is loaded at runtime needs to have a specific implementation. The Reality Editor would then basically require the implementation of a content management system similar to what a web browser already provides. The first experiments with an openFrameworks-based server also made clear that most of the user interfaces do not require any special 3D engine effects, since most of the interfaces only require two-dimensional graphics. Since it was the goal for the Reality Editor to generate stable objects that could potentially last a significant amount of time, Unity was not a viable option. Instead, it became more and more clear that the World Wide Web could provide all of the benefits required for the Reality Editor to work.
The resulting experiment focused on the potential use of ordinary HTML code to be transformed into 3D and used with augmented reality. The first idea was to generate web views via objective-C and transform them natively. However, this did not work very well, since the communication among the interfaces and the Reality Editor needed extra native implementations. A follow-up experiment focused on the feasibility of injecting the AR tracking transformation matrices from the native app into the web view and using these matrices to trigger 3D transformations of HTML elements. The successful results of these experiments are described in chapter 4.5, Augmented Reality Web. Now that it was feasible to use CSS 3D transformations with HTML elements, the question emerged how it would be possible to inject user interfaces into a single webpage that would 3D-transform the content. The first ideas were to use AJAX to load content. However, every idea led to problems when considering independent content from unknown sources. Eventually the idea was formed to build a system that would generate iFrames just-in-time, as needed, and load the augmented reality content into these iFrames.

This concept represents the final implementation for the Reality Editor user interface. Once a new marker is found, the remote webpage associated with the marker is loaded into a just-in-time generated iFrame and 3D transformed using CSS-3D transforms. Once the marker and therefore iFrame are no longer visible, the Reality Editor starts a counter and destroys the generated iFrame after three seconds. This way, the Reality Editor user interface stays agile.

At the point where the iFrame system was built, many calculations and preparations were still programmed in native code, but the idea was to build a stable tool on web technology. Therefore, with a new experiment, I pushed as many calculations and preparations as possible into the JavaScript code of the webpage. At the time these experiments took place, the iOS platform was matured enough to provide sufficient processing resources for such experiments. Additionally, the WebKit JavaScript engine in iOS was already very strong. The experiments did not show significant differences when processing the 3D calculations in JavaScript compared to processing them natively. Eventually the only code that stayed in the native application was code that just cannot be executed in HTML. This is for the discovery of new objects, the downloading and activation of new trackable marker information, and the AR tracking system itself.
Another path of experiments was the exploration for the right server foundation. It was clear after the initial Reality Editor experiments that there need to be a distributed system. The centralized system already would implement the simple protocol described in chapter 4.3, Unified Data. However, a distributed system was required to really enable a scalable number of objects to interact with each other independently from a central master plan. Initial experiments were written in C for Atmel ARM processors and the CC3000 Wi-Fi chip. A Lego car with Mindstorm sensors and motors was built and connected to an ARM MCU that would communicate via HTTP with the Reality Editor. It became very difficult to program the HTTP service for real-time communication via C. It also seemed too specific and it became impossible to extend the server implementation to other systems. Additional the CC3000 did not seem capable enough to allow a distributed system on the scale needed for the Reality Editor.

At the time of these experiments, a System on a Chip (SoC) capable of running Linux became very affordable, with a retail price of around $35. The idea was formed that by the time the Reality Editor research was finished, such systems would enter the $2 price range and therefore become a feasible solution for distributed IoT objects. A benefit of a full embedded system is that an entire complex service could be run without the need of any centralized unit. The first prototyping hardware platform for these experiments was the Arduino Yun. The Arduino Yun combines an Atmel Atmega processor with a SoC from Qualcomm on one board. Both systems are bridged via a serial interface. The idea was to simplify the development of connected objects and therefore Node.js was chosen to program the entire distributed service infrastructure. Because Node.js is JavaScript, it allows a single developer to work on the Reality Editor user interface as well as the service. Node.js also has the advantage for the Reality Editor that it is an asynchronous event-based programming language. The Reality Editor service has to deal with real-time data signals sent around a distributed network. The system needs to permanently act on incoming events and then execute a cascade of related events. The concept of asynchronous callbacks instead of procedural program loops represents the concept of the Reality Editor.

In the beginning, the Reality Editor service was focused on providing a simple functionality, but over time it grew into a multitude of functionalities that needed modularization. The Reality Editor server would first get a modular plugin system for the hardware interfaces and later the
The concept of modularization would enter every aspect of the service. Eventually the modularization enabled logic crafting, as each logic block forms an independent module loaded as a plugin into the Reality Editor server and from there exposed to the Reality Editor.

4.1. Server

![Diagram for the architecture of the Reality Editor server.](image)

The Reality Editor server makes use of a few internet and web technologies. It uses HTTP REST to communicate with the Reality Editor, and WebSockets to exchange real-time data among connected objects and the Reality Editor. UDP broadcasting is used to establish a local decentralized network. Decentralized in this context means that the system acts as if the network is decentralized, although this is physically not the case when operated in a private home network with a single Wi-Fi router.

Within the private network, all objects discover each other using UDP broadcasting heartbeat messages sent out at regular basis. These messages consist of a connected object’s name and the IP address where the object can be reached. Once two connected objects or an object and the
Reality Editor know about each other, the communication continues on the basis of HTTP REST and WebSockets. To keep the network up to date, all objects additionally react to so-called action messages broadcast via UDP.

All connected objects autonomously contain every aspect of their being. From a data perspective, all connected objects are self-explaining. This means that every connected object stores its own AR-target information, its HTML interface data and all data about its state and logical operations. Each connected object stores a central database in the form of a JSON object, which holds all states and data used by the connected object. At the highest level this object stores essential information such as network IP address, a unique ID, and 3D positioning data for the AR-GUI.

This JSON object also holds information about all connected links and existing nodes. All nodes hold a data object that stores the real-time data value for the node. If a node is a logic node, the node encapsulates a second list of links and blocks. The blocks also store data values. All links contain origin and destination information. All nodes contain information about their 3D position as well as configuration information. This structure is particularly beneficial when dealing with distributed data. The Reality Editor can easily load the entire JSON object to understand how to render the user interfaces. On the other hand, a logic node consists entirely of the information stored in its node. If a user wants to copy a logic node, the Reality Editor only needs to store an instance of the logic node. The processing blocks for the logic nodes are stored separately in the modular plugin structure of the Reality Editor server. The central JSON object helps keep all components of the server synchronized.

The Reality Editor server keeps an extra data object for storing other known connected objects as well as data for how to access these objects. For example, the latest version of the Reality Editor server modularizes the communication protocol. This means that objects could work on different protocols yet own protocol modules, in effect speaking different dialects with each other. The interface content is provided with a static web server that delivers HTML, CSS and JS files stored in the Linux file system. Each time an important data point is changed, such as when a link is established or deleted, or changes are made to the AR GUI 3D positioning data, and a full
copy of the central JSON database object is stored in the Linux file system. This copy is loaded at the system start to reestablish the connected object’s behavior.

For enabling the creation of new hardware interfaces or new connected objects, a simple API was created. This API is used to connect new hardware interface modules to the Reality Editor Server. At this time, there are plugins created with this API for the Lego Mindstorms EV3, Lego WeDo 2.0, Philips Hue lights, the Raspberry PI, and the entire Arduino product range, as well as certain webcam products. The Arduino interface, however, comes with a unique set of functionality. Beside a serial protocol that would connect a computer or a SoC with the Arduino board via a serial connection, a library for Arduino was written that extends the Node.js API into the Arduino programming language. The Arduino library mainly translates between function calls in both environments and handles the communication with the Reality Editor server. Only four function calls are required for establishing a new nodes and data communication. A developer can thus establish developer functionality, add a new node, write and read data from the server.

The Reality Editor consists of four main event receiver units as well as four complementary sending units (Figure 13). A beatServer unit is listening to broadcasting messages sent out by other connected objects via the beatSender unit which sends out name and IP address of each connected object. The received heartbeats are compared with the stored database of known objects. If a new connected object is found, it is added to the database. The actionSender sends messages similar to the heartbeat when the entire system needs to act on a special event. This could be when multiple user interfaces have to operate at the same time and the entire system has to update frequently or a new UI requests instant responds for heartbeats. A socketServer handles all incoming Web-Socket requests for real time data transfers, while a socketSender actively pushes data to other connected objects. The socketServer has a helper unit called socketUpdater, which makes sure that all possible WebSocket connections are open and alive at any given time. A webServer serves the AR-GUI via HTTP. Finally, the hardware interface module provides the communication with the hardware.
When the socketServer or the hardware interface module unit receives a new data value, meaning a node event occurred, the object engine searches through all stored links to check whether the changed node is the origin for any link. If a link with the origin node is found, the object engine processes the link. If the link destination is external, the data is sent via the socketSender to another connected object. If the link destination points to a node within the Reality Editor server, the data is processed internally. If the destination node is linked with a hardware interface, the data is processed by the hardware interface and sent to all event listeners that established via the hardware interface API. In case the node is handled within the Reality Editor server, the object engine is called again to process the destination node as a new origin node. Such object engine processes are happening as long as links show dependencies on changed origin nodes. By means of data IDs, the object engine prevents internal and external endless loops, which would stop the server from operating. In those cases where the data is sent to a logic node, the object engine runs a separate process with the logic node blocks and links. This second process sends the data to the logic block modules to process the incoming data. Once the block modules are processed, the data is given back to the normal node object engine process. Since each logic block can potentially have four data points, but every normal node only processes a single data value, a special translation block is used between the input and the output of the two object engine processes.

4.1.1. Data communication format
The Reality Editor saves all data values in a specific JSON data format. This format is used internally to process the data values and externally to send the data values from one object to another object. When sent to another object, the data package is combined with additional addressing identifiers. A data package that is sent from an origin to a destination object has the following JSON data:

- **object**: Represents the unique identifier of an object.
- **node**: Represents the unique identifier of the destination node. Usually it is the name of the node combined with the object identifier.
- **data**: Represents a data object describing the transmitted number value.
The Data object comprises:

- **value**: Represents the actual number sent.
- **mode**: Defines the kind of data sent. At this point three active data modes are implemented.
  - Mode (f) defines floating-point values between 0 and 1. This is the default value.
  - Mode (d) defines a digital value exactly 0 or 1.
  - Mode (+) defines a positive step with a floating-point value for compatibility.
  - Mode (-) defines a negative step with a floating-point value for compatibility.
- **Unit**: Defines a string of the name for the unit used (for example "C", "F", "cm"). Default is set to no unit.
- **unitMin** and **unitMax** defines the scale of the unit that is used. Usually the scale is between 0 and 1.

A final example JSON data package looks as followed:

```json
{object:"objectABC222dddd88", node:"switch", data: {value:0.5, mode:"f", unit:"", unitMin:0, unitMax:1}}
```

### 4.2. Reality Editor

When the Reality Editor (Figure 14) is started the web view interface loads the present state of all connected objects found in the local network via the UDP heartbeat messages. For this purpose, the heartbeats are send from the native app into the web view interface. The web view interface downloads all requir JSON data from the remote object via HTTPs. Cross-Origin Resource Sharing (CORS) HTTP standard is used for this download in order to bypass the same-origin limitation that usually applies to resources loaded into a web page. Additionally, the native app loads all AR tracking data from the objects, so that they can be visually discovered.

Once the native app and the web view interface have gathered all necessary data, and the connected objects are visually tracked, the web view interface loads the AR-GUI via HTTP from the connected object. These AR-GUIs are loaded into iFrames that are generated just in time, so that each AR-GUI can operate independently. As soon as an AR-GUI is fully loaded, a WebSocket communication between the AR-GUI and the connected object is established. This
allows the AR-GUI real-time interaction with the connected object. The JavaScript library Socket.io is used for this purpose. When a user draws a line from an origin node to a destination node using the crafting interface, the UI uses HTTP REST methods to establish (POST) a new link within the origin connected object, as well as to delete (DELETE) such a link. A link tells the origin connected object where it should send processed data signals. As a result, connected objects communicate among themselves via push messages. Once a link is established, the origin connected object keeps the WebSocket connection with the destination connected object alive at all times. This guarantees real-time data communication. All objects communicate directly with one another. At no point is there an Internet connection or a cloud service involved, hence the Reality Editor platform can be considered decentralized.

![Diagram for the architecture of the Reality Editor mobile app.](image)

The native part of the Reality Editor is programmed for the iOS operating system. openFrameworks [47] is used to keep the code easily platform independent. This combination
provided a very simple workflow and a simple use of the Vuforia framework facilitated with an addon [52]. The web view interface is established with native objective C. As described earlier a maximum of code is pushed into the web view interface for generating maximum compatibility in case the platform is ported to another system. The native app comprises a UDP broadcasting discovery unit (discovery objects) and an AR tracking unit that is combined with an AR target download unit. The AR target download unit adds new targets to the AR tracker at runtime. The native app can call JavaScript functions within the WebKit interface. The app uses this function call to provide the web view interface with all necessary data (addObjectFromBeat) so that the GUI can load all the data needed for directly interacting with a connected object (getObjectData). Once this data is loaded another function call is used to send the tracking data via a setProjectionMatrix unit to the Web View Interface GUI 30 times a second. The GUI uses this function call to process all 3D data accordingly to the requirements of CSS3 3D-transformations. With the same function call, it also loads new AR-GUI elements from connected objects into iFrames or it destroys iFrames once the connected objects have not been visible for a certain time. Finally, GUI elements are composed from a drawAllLine unit that provides the programming lines and a DrawInterface unit that provides additional interfaces such as 2D buttons and setup functionality. Eventually, all of these elements and changes are written into the DOM so that the GUI can be updated. For interaction purposes, touch event listeners are used to animate the programming lines and establish or delete the programming links.

4.2.1. **OpenGL to CSS3 Matrix3D**

The Reality Editor uses a novel method to 3D transform HTML webpages. Once I determined that 3D transformations are hardware accelerated, the challenge was to figure out how to combine OpenGL with the CSS3 transformation pipeline. There is not a standard method to convert OpenGL transformations into accurate CSS3 transformations.

The OpenGL rendering pipeline and its space coordination system is the common way for augmented reality tracking systems to transform 3D graphics in space. It exposes to the developer a projection matrix and a model-view matrix. The projection matrix provides information about the camera properties, while the model view matrix supplies information about the position of a point in 3D space. Multiplying both matrices produces the right
positioning for 3D points to be rendered onscreen. However, the CSS3 transformation matrix does not provide a projection matrix. Therefore, the research for a method to transform one space into the other was required.

![Diagram of OpenGL rendering pipeline]

*Figure 15 An illustration showing the OpenGL rendering pipeline.*

When the augmented reality tracker detects the position of a marker, the tracker provides a model-view and projection matrix, both in the shape of a OpenGL conform 4x4 matrix. After the initial vector of a 3D object is multiplied with the model-view matrix and the projection matrix, the resulting coordinates are divided by the screen coordinates \((w')\) to be normalized and then scaled to the viewport dimensions. Eventually the scene is rendered on the screen (Figure 15).

![Diagram of WebKit CSS3D transformation pipeline]

*Figure 16 An illustration showing the WebKit CSS3D transformation pipeline.*

The WebKit rendering pipeline operates similarly to OpenGL’s rendering pipeline. However, there are several key differences (Figure 16). For one, a transform matrix can be set by setting the `-webkit-transform3D` property of a HTML tag. This setting can be compared to the model-view matrix in OpenGL. There is a problem with this comparison, because the projection matrix cannot be set manually, i.e. by defining the frustum or by setting each of the 16 elements of the
4×4 projection matrix. As a workaround, the -webkit-perspective3D property of the DIV can be used to manipulate the projection by using a single distance parameter. However, this method does not correspond to the OpenGL projection matrix. A perspective divide that normalizes the openGL coordinates to the screen coordinates is not possible. Another method had to be found.

4.2.2. The OpenGL-CSS3 Transformation Matrix

![Diagram showing the WebGL CSS3D rendering pipeline modified to match the openGL pipeline.](image)

*Figure 17 An illustration showing the WebKit CSS3D rendering pipeline modified to match the openGL pipeline.*

Keeping in mind the differences between the two described pipelines, it is possible to build a transform matrix that accounts for all the differences (Figure 17). As it appears that the viewport for WebKit’s rendering is the DIV itself, it is requiring to first set the transformed DIV’s dimensions to be set as equal to the screen’s dimensions. In the used implementation, the iFrames used to host content are embedded in the div that requires to be in the size of the screen coordinates. The 4×4 matrix that transforms the DIV can be built by multiplying the following matrices in order:

1. The ScaleXY matrix:

   
   \[
   \begin{bmatrix}
   \text{scaleX} & 0 & 0 & 0 \\
   0 & -\text{scaleY} & 0 & 0 \\
   0 & 0 & 1 & 0 \\
   0 & 0 & 0 & 1
   \end{bmatrix}
   \]
As a first step the X and Y coordinates are scaled. ScaleX and scaleY represents the desired scaling factor of the HTML interface in the x and y-coordinates respectively.

2. The model-view matrix is just OpenGL’s model-view matrix, which can be retrieved from the AR Tracker.

3. The ScaleZ matrix.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

An abnormality in the z-positioning of the interface was observed, and solved with an additional matrix multiplication. The factor of 2 at position 11 was a result of experiments.

4. The projection matrix is OpenGL’s projection matrix, which can be retrieved from the AR tracker.

5. The ViewportScaling matrix.

\[
\begin{bmatrix}
\frac{W}{2} & 0 & 0 & 0 \\
0 & -\frac{H}{2} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The projection matrix used transforms the coordinates into normalized screen coordinates and there no viewport scaling occurs afterward, so the scaling needs to happen here. The factor of \( \frac{1}{2} \) was used because the fact the normalized screen coordinates range from -1 to 1: a span of 2.
Therefore, we only need to multiply the normalized screen coordinates with half of the screen’s dimension (W = screen width, H = screen height) to place the pixels correctly on the screen.

Multiplying the five matrices above in order and using the result as the value for the -webkit-transform3D property of the interface lead to a successful combination of OpenGL coordinates with CSS3 3D transforms.

### 4.2.3. iFrame transformation

The Reality Editor in its present form implements an optimized version of the transformations described above. The matrix multiplications are minimized to only those that permanently change, and additional calculations are added to provide additional repositioning of user interfaces.

Since there is only a fixed camera property coming from the AR tracker, the scaleZ, projection and viewportScaling matrices can be pre-calculated. Experiments have shown that these calculations are not fully consistent among all devices. Since the camera and the screen coordinates of each smartphone differ minimally, the viewportScaling is complemented with x and y calibration coordinates at positions 13 and 14.

For each frame, the resulting matrix is multiplied by the model-view matrix. The resulting matrix is used by the Reality Editor to transform all iFrames that are attached to a tracked marker. For each such iFrame, a first invisible DIV container whose screen size exactly fulfills the requirements of the method used. The actual iFrame is positioned at the center of the DIV container, so that the 3D transformations applied to the DIV container transform the iFrame in the same way. The content of the iFrame sends an HTML postMessage to the Reality Editor to provide the accurate size of the content, which is then used to modify the scale and position. An additional transformation matrix is added, representing the exact position of the iFrame user interface, relative to the center of the tracked marker. Eventually, the resulting transformation matrix is applied to the DIV container.
4.3. The Influence of Object Identification on Network Infrastructure

A question remains regarding how the Reality Editor can identify connected objects and how these methods influence the network communication. To enable precise identification and augmented reality tracking for a connected object, a two-step process is required. In the first step, the Reality Editor needs to understand the computational and visual identification of a connected object. This means that the Reality Editor needs to know enough detail about the visual appearance of a connected object, as well as a unique identifier and the network IP address where the Reality Editor can communicate with it. There are different methods to acquire these three data components. Each method enables a different networking infrastructure and use for the Reality Editor.

The networking system used in this thesis relies on a distributed networking infrastructure, where the last edge node is a home router that enables local broadcasting via UDP. In such a network, it is easy to communicate the relevant information between connected objects and the Reality Editor. Once the Reality Editor is opened, it sends a broadcast request into the network, which all of the connected objects receive at the same time. The connected objects respond with a broadcast message that contains their network IP address as well as a unique identifier (UUID). The Reality Editor uses the IP address to request a visual reference for tracking the connected object. The Reality Editor saves the visual reference in combination with the UUID and the IP address. From this moment on, if the Reality Editor detects the connected object via the camera stream, it can use the saved UUID and IP address to request the user interfaces.

This method is dependent on infrastructure that allows UDP broadcasting. While most personal Wi-Fi has this capacity, it becomes more difficult when the network has restrictions on broadcasting or if it is part of a bigger network infrastructure, such as a corporate or university network.
In cases where broadcasting is not applicable another method needs to be found. The Internet Domain Name System (DNS) is a good example. Domain Name Servers translate domain names into IP addresses, by providing lookup databases that allow browsers to match a domain name with an IP address. The IP address is then used to communicate with the server. The DNS service is distributed, meaning that many DNS servers exist and they synchronize data with a centralized root name server. The Reality Editor can use a similar system, called a global target system (GTS). GTS saves a reference for UUIDs, IP addresses and physical global locations that let it find a connected object or an AR experience. The Reality Editor uses these GTS services when a user wants to engage with a connected object that is on a separate network, such as one located some distance away, or on a network that has no broadcasting enabled.

For GTS to work with the Reality Editor, the target connected object or AR experience has to be registered with the service first. This means that the location of the target as well as an IP address from which the relevant identification and interface data can be downloaded has to be registered with the service. Once the Reality Editor starts, it sends a request to a list of known GTS services. The request includes the physical position of the Reality Editor. The GTS service looks up all possible targets that physically surround the Reality Editor, starting with the closest ones, then moving farther away. It searches until the maximum limit of targets has been reached. Currently, the limit is set to 200 targets. Once a target is found in the GTS database, the IP address associated with that target is sent to the Reality Editor, and the Reality Editor uses it to download the visual identifier information and UUID. From that moment on, the system behaves exactly as it would with a local broadcasting system.

Connected objects can also use GTS to find nearby connected objects and use the Internet for interconnections. However, this option has not yet been enabled, since additional security requirements need to be considered before building private networks of things using the public Internet.

While the local broadcasting mechanism can easily be scaled up in individual homes, the GTS approach can be scaled worldwide using Internet infrastructure. This is particularly useful where AR experiences are enabled with objects that are always in the same location, such as retail
stores, city infrastructure, or stadiums. For example, a stadium owner can use GTS to register AR experiences inside or around the stadium, using a normal web server. A user never needs to log into a local Wi-Fi network, but can use a mobile Internet connection to request targets and interact with the AR content in the physical environment. An advantage of this system is that AR experiences can be built on common Internet technology owned by the content provider. The GTS service can be operated by everyone and saved in the Reality Editor’s internal lookup database. However, when it comes to objects that can change their location, such as cars, toys, or other household goods, the GTS service falls short, because it is always dependent on a having a fixed registered location to find the target.

4.4. Alternative Object Identification Solution for Decentralized Augmented Reality Applications

To solve the problems occurring with these two methods, I researched a third method. The main problem with both approaches was that there can be no true decentralization as long as the initialization requires a third party to map a visual identifier with a content server. The only satisfactory solution was to find a way for the connected object or AR experience to provide the mapping information itself. As such, a connected object would need to store a reference to the IP address where its content and UUID can be requested.

There are possible solutions available already, with the most recognizable technologies being radio frequency identification (RFID) tags [53], quick response (QR) codes [54] and barcodes [55]. However, each of these technologies comes with a particular disadvantage for the specified use case. The ideal solution needs to communicate to the user that AR content is to be expected, and also be attractive to the human eye. QR codes and barcodes were designed for computer readability, not aesthetic appeal. There is research on making QR codes more attractive [56][57][58], but at its core, the QR code appears chaotic to the human eye. RFID tags can communicate relevant information invisibly, but come with another set of problems. Precisely because they are invisible, it is difficult for them to tell users when to expect interaction [59].
Although it is possible to combine RFID with a clear visual target identifier that could signal that augmented reality content is to be expected, this combination of technology is difficult to realize with today’s mobile phone infrastructure.

A possible solution is a visual identifier that serves the purpose of a QR code, communicating its use to a human and initialization data to the Reality Editor, but that also appeals to the human eye as a graphical element, preferably with an ornamental style that the user would be willing to place out in the open. The solution should blend into the fabric of the physical world and connected objects, yet provide enough identity to communicate that an AR experience is to be expected when the user points a tool such as the Reality Editor at it. Such a visually interesting solution would enable a decentralized AR system independent from third parties. This means that such a code or ornament could hold information about a remote server owned by the connected object manufacturer, which would provide the Reality Editor with all of the information it requires to establish the AR experience.

Research on possible solutions to the challenge of designing codes or ornaments that would solve the problem found a variety of pixelated ornamental geometric shapes traditionally used to convey messages and provide a commonly recognizable style that communicates a dedicated purpose. For example, Chinese seals were used to mark paintings [60] (Figure 18/1), and ancient square Kufic calligraphy is still used for logos of institutions and businesses around the world [61] (Figure 18/2). These ornaments inspired experiments that led to a human- and machine-readable, QR code-like ornament.

For centuries, pixelated ornaments were used to decorate the walls of buildings. In some cases, these decorative ornaments also contained messages [62]. Examples of these kinds of ornaments are found on the mausoleum of Tughluq Temur, erected in China in 1363 CE, and the Jameh

![Figure 18](image-url)
Mosque of Isfahan, erected in Iran in 771 CE. Both are decorated in a calligraphic style called square Kufic, the creation of which follows a very strict geometry and algorithm [63]. The abstraction of the writing is so high that reading it at first glance is not possible. This is done intentionally, so that the writing becomes truly decorative.

Throughout the history of calligraphy, there have been many influential cultural exchanges between the East and the West. The printing press had a major influence on Arabic writing and the further development of square Kufic calligraphy [62]. In addition, the Latin alphabet was used in combination with square Kufic styles for business signs in Turkey [64]. A modern interpretation of Kufic by Lina Abdul Hadi, called Kufin [65], uses modern typeset software that allows for the combination of certain letters. As these typesets are limited in their ability to morph and dynamically change characters, these tools do not allow for flexible changes to the individual letters based on constantly changing letter combinations. The resulting typography does not look fully ornamental, nor does it blend into the background. Interesting inspiration can also be drawn from the marks of Chinese authors found on many ancient and contemporary drawings [60] (Figure 18/1). Mamoun Sakkal points out that there is an inspirational link between square Kufic and these earlier Chinese drawings [62]. Kufic even seems to be a source of inspiration for the constructivism of artists around the De Stijl movement and the Bauhaus school. The work of Piet Mondrian [62][64] is an exceptional example.

Of course, there are many other visual codes that allow a user to point a scanning device at a target and instantaneously decode its hidden message (Figure 18/3). These machine-readable codes can either contain complex messages or simply hold a numeric pointer to a message in a clearinghouse database. For example, Data Matrix [66] and QR codes [54] can contain full messages within them, whereas CyberCode [67], the TRIP system [68] and barcodes [55] only provide a pointer to a message. All of these codes share a similar visual approach. Black and white contrast is used to encode bits into a visual representation. The placement of the black dots is optimized for a computer system; therefore, these dots appear chaotic and random to the human eye.
There are also so-called visual fiducial markers (Figure 18/4) that look more appealing, but these markers do not encode information. Rather, the overall appearance of the marker generates a unique ID (UUID) that then can be used with a system that has information associated with the UUID assign an action to the fiducial markers. One example is d-touch [69], a system that allows for different shapes and appearances that are detectable even when the markers are distorted. Many other solutions are similar or build upon the d-touch technology [70][71][72][73][74]. In the field of augmented reality we can see a further development of such fiducial marker with so-called natural feature tracking, where for example scale and rotation invariant features of known image markers can be detected in a video stream [75]. Fiducial and natural feature markers have in common that a reference or pointer ID has to be previously known in order to evaluate the marker.

Looking at all this inspiration, it seems like a possible solution could be a combination of some sort of matrix code such as QR codes with ideas of Kufic, constructivism, and Chinese seals. Fiducial markers will not be able to generate codes that contain information that points clearly to identifiers.

Figure 19 An ornamental alphabet resembling the Latin alphabet with each letter being represented in a 3x7 pixel grid.
The idea that resulted from this research was to come up with an encoding of the Latin alphabet that would be easily machine decodable. The letters should have such a degree of abstraction that when combined into a single flow of connected shapes, they would be hard to read and identified as an ornament. Figure 19 shows the resulting alphabet. Inspired by the pixel encoding of Kufic, each letter is created in a 3x7 pixel grid and messages are written from left to right, row after row. The following rules were defined to form ornamental shapes.

"I" Each letter consists of 21 pixels, seven in height and three in width.

"I am" If letters are connected with a dot, they form a word. (The dot is red here for illustration purposes.)

"I can" If the word continues in the next row, the last letter is connected with a dot to the row below.

"I can" Messages are terminated with the sign marked in red. The rest is a three-letter checksum, followed by random letters to fill the shape of a square.

"I can" Every square has twice as many letter rows as columns.

Based on this rule set, these human-readable quick response (HRQR) codes interweave letters to fully occupy the available space and form a homogeneous ornament. In order to generate an ornamental line flow, the letters forming a word are connected with a single pixel placed in the empty pixel column between the letters. Since this pixel greatly contributes to the ornamental flow, a special algorithm was developed that places the connecting dot at the best visual position.
To find this optimal position, every pixel outline that can connect to another letter is evaluated for best connectivity (Figure 20).

![Figure 20](image.png)

*Figure 20 Visual evaluations of letters for placing a connecting dot between the letters.*

The outer border of each character is given a score that represents the ideal connection points. A score of one represents the best possible connection, where the pixel is facing the end of a line that can be elongated. A score of two represents a corner of a shape that would generate an acceptable visual appearance. A score of three represents a pixel that is part of a line that would not generate a satisfying flow. A score of nine represents an empty pixel that cannot be connected.

Based on this evaluation, the algorithm described in Figure 21 is used to calculate the best connection positions between two characters. The lowest resulting score from top to bottom represents the best visual position for the connecting dot. If the letter is at the edge of a square, such that no more letters follow in the same row, the dot connects to the letter in the row below.

![Figure 21](image.png)

*Figure 21 Algorithm for evaluating the best connection position between two letters.*

In order to fulfill the set of rules described earlier, the resulting ornament must always form a square. However, not every message has the optimal length for filling a full square. Therefore,
the rest of the square needs to be filled with random letters. In addition to the message and the randomly generated letters, a checksum is included in the ornament using a 16-bit cyclic redundancy check (CRC) algorithm [76]. The checksum is defined to always fill the space of three letters. The checksum block of three letters is located between the message and the randomly generated letters. A special letter is used to mark this separation. A resulting ornament with visually marked separations can be seen in Figure 22.

![Figure 22 The HRQR message “Human Readable Quick Response Code,” with visually separated components.](image)

4.5. Computational Decoding

This section describes one version of an algorithmic pipeline for decoding the ornamental encoded messages described above on a mobile device.

An image taken of a planar ornamental code can have any possible perspective transformations and rotations. As a first step, the software needs to determine the exact position of the square tag in a given camera image, such that the ornamental code can be processed. The following steps need to be performed (Figure 23):

For computational efficiency, the image is first transformed to grayscale (Figure 23/1). As the ornamental code always uses the same space between white and dark pixels, a morphological erosion filter is used to extend the dark pixels beyond their borders. The resulting image shows a closed black square (Figure 23/2).
Blob and contour detection is used to find all closed contours in the image (Figure 23/3). The resulting blob contours consist of many points forming the outline of the blob. Beside the real edges of the shapes, these points also represent false detections along a straight line (Figure 23/4). As we are only interested in real edges and therefore lines that form the outline of a square, we remove all points that are not part of strong edges (Figure 23/5). The software sorts all corner points, such that the longest lines of the square can be found. As not all letters have perfect black corners, we extend the longest lines to the edge of the recorded image (Figure 23/6). The intersection of these extended lines found within the borders of the recorded image form the exact corner points of the shape (Figure 23/7). If the shape has four corner points and these four corner points could potentially relate to a square, the shape is isolated from all other shapes. The possibility of the shape being a square is calculated with \((a+c)/(b+d)\) and \((\alpha+\gamma)/(\beta+\delta)\), where \(a, b, c,\) and \(d\) are the side lengths of the shape, and \(\alpha, \beta, \gamma,\) and \(\delta\) the corner angles of the shape. The closer these ratios are to 1, the more likely it is that a square has been found. A threshold is used for deciding if the value is close enough to form a square or if the software should continue to the next shape or frame (Figure 23/8). Once the corner points of the square are found, a homography \([77]\) is calculated. The square shape is then rectified using a perspective transformation for further processing.
Before the message in the ornamental code can be read, the perspective-rectified shape needs to be analyzed in order to estimate its correct orientation and grid size. We can analyze the detected square with the following rule set: Every row is only separated by a maximum of one dot, and most likely every column has multiple dots connecting letters. The number of columns is always double the number of rows. To determine the correct orientation, we can use the mean value for each pixel row and pixel column to test against these rules. The rows and columns mean value graph (Figure 24) shows a clean image of how the ornamental code is oriented in our perspective-rectified image.

The graph in Figure 24 is taken from the example ornamental code used for illustrating Figure 23. Since the rows should have only a maximum of one connecting dot, the maximum mean value (green) should be higher than the maximum mean value of the columns. One can see that the row graph has the higher maximum values. Additionally, the number of rows should be half the number of columns, which also can be seen in the row graph. Subtracting 10 from maximum peak (yellow line) gives us 18 line intersections for the column graph crossing the yellow threshold. In contrast, the row graph only shows 8 line intersections. Combining the findings of the highest peak with the count of lines, it is most likely that the row graph represents the rows. If the ornamental code would show the wrong orientation, the image is rotated 90 degrees (Figure 25 Step A).

The previous calculations also provide the exact number of rows as well as the exact offset of the ornamental code by showing the amount of white space outlining the ornamental code (red and purple). This is a result of the erosion filter used earlier. Since we know that each ornamental
code has twice the number of columns as rows, and each letter is 7×3 pixels, we can now segment the ornamental code into its rows and columns (Figure 25 Step B) and use the line pitch of each column as well as the offset to calculate a pixel grid that matches the pixel grid of the original ornamental code (Figure 25 Step C).

![Step A](image1)

![Step B](image2)

![Step C](image3)

![Step D](image4)

**Figure 25 Determining orientation and segmenting the ornamental code.**

At this point, it is already possible to start reading the message encoded in the ornamental code. However, only the horizontal orientation has been confirmed so far. The image could still be upside down. In order to get the final correct orientation, the separating special character for the message block is useful. Since this character is always present in an ornamental code, it can be used to verify the right orientation. In an initial fast search, the software looks for the special character only in both flipped and normal orientation. (The algorithm performing this search is described in the next paragraph.) If the sign is not found, the software stops all additional calculations and continues to the next image frame. If the sign is found flipped, the image is rotated 180° (Figure 25 Step C and D). Eventually the right orientation is found and the text can be decoded.

The value of each letter pixel can be found by extracting a sample pixel at the center of each letter pixel in the grid. The measured values are binarized using a thresholding operation. Every
letter contains exactly 21 (7x3) pixels and therefore 21 bits. As a result, every letter can be stored as a long integer (32 bits). With a lookup table that stores all letters encoded in long integer numbers, the text decoder loops through the lookup table to search for an integer number matching the one of the present letter (Figure 26). If a pixel and therefore a bit between two letters is found, the letters are connected to words. The message, checksum and random letters blocks are separated by searching for the special character sign. The search is safely performed backwards, as the checksum and the random generated letters do not contain any letter similar to the special character. Eventually the 16-bit CRC checksum of the message is calculated and compared with the checksum encoded in the message.

Figure 26 Screenshot from a prototype that shows the square detection, the mean value calculation, the matrix detection and the resulting text decoding.

4.6. Ornamental Codes for Augmented Reality

The described algorithms for encoding and decoding showed a proof of concept for how a matrix code can be generated to support the needs of a decentralized AR marker. However, they still
miss additional steps to support the needs of augmented reality. For one, the approach is not stable for occlusions of any part of the code. Additionally, the square detection has many false positives and is not optimized for tracking in all possible camera perspectives. However, it is possible to use the detected code and hand over a reference image of such a code to a AR-tracking engine like the ARToolKit framework. As such, the ornamental code can be used as an AR marker that can encode a URL to a remote server independent from a location reference. Therefore, the ornamental codes represent an additional method for AR tracking.

With help from Eyþór Rúnar Eiríksson, additional methods for tracking the ornamental codes were explored. Eiríksson made use of work described by Dubská et al. [96][94], where an edge response image is computed and a Hough line transform of the edges is performed. The resulting space is analyzed to detect two groups of parallel lines that have dominant vanishing points. A grid is constructed and the image sampled at each line intersection. Eiríksson was able to implement this new method and perform template matching for each defined symbol on the sampled binary image with a framerate of 20fps. As a result, every character could be detected as individual visual object and as such the detection would become very stable. Eiríksson used the 3D position of the marker in its own reference frame and the camera intrinsic parameters to solve the perspective-n-point problem [78] and thus estimate the marker’s pose in the image [79]. Thus, this exploration by Eiríksson allowed a stable tracking of ornamental codes for augmented reality purposes.
5. Interface Functionality

The previous chapters described the core functionality and the system implementation for the Reality Editor. This chapter describes the user interfaces that were implemented in the Reality Editor to provide additional tools and a better user experience. Significant time was spent to implement a simple user interface that is optimized for augmented reality and empowers the user to understand and edit connected objects. There are visual tools that allow a user to create augmented reality experiences. A memory system allows a user to seamlessly jump among connected objects even when they are not physically present at the same location. A method allows for locking connected objects so that connections and settings become personalized to a master user. Most of the described interfaces use the visual camera stream for interacting with objects, while a second method called instant connection allows objects to be connected without a visual reference. Finally, a visual programming language is implemented and described in this chapter that allows users to control connected objects and the interactions between them using small logical programs placed in augmented space.

5.1. The Main Interface Buttons

Five essential functionalities grew from the Reality Editor research. Each of these functions is represented with a transparent button on the right corner of the Reality Editor. The buttons are transparent to expose as much video screen as possible, for drawing the augmented user interfaces. The first button exposes user interfaces for interacting with a single object, the second button exposes an interface that allows for connecting and programming objects. The third button represents a virtual pocket that allows a user to place virtual interface elements on to augmented connected objects. A settings button opens a setting menu for configuring the Reality Editor. The last button allows the video stream to be frozen. All directly mapped user interfaces active on screen are frozen in position, yet stay active for interactions.
5.1.1. Interacting with a single Object

![Figure 27](image)

*Figure 27 A screen capture of the Reality Editor showing social media buttons.*

The first method of interaction is the ability to interact with a single object. This means that an augmented user interface allows the manipulation of the object a user is focusing on. These interfaces are usually webpages, which a user can create. The infrastructure for these user interfaces is similar to that of regular web services. Therefore, every known technology developed for the World Wide Web can be used to generate such an interface (Figure 27).

For example, JavaScript can be used to establish live communication between a remote server or an object and the Reality Editor interface. A company that serves millions of clients with an interactive webpage can reuse the same infrastructure to provide the same content for augmented reality using the Reality Editor.

5.1.2. Connecting Objects

The second user interface method enables making connections between objects. As previously described, the Reality Editor allows the breakdown of an object into its essential components. The user interface for connecting objects exposes these components as visual nodes. Each node is a predefined webpage that uses the same iFrame technology as interactive webpages. However, the nodes are preprogrammed into the Reality Editor user interface, so that all objects show the same consistent interfaces. Using this mode, a user can draw a line from one node to a second node to establish a connection. Once the user’s finger is released from the screen, the Reality Editor sends a link definition message to the object belonging to the first touched node. This definition message tells the object that from now on, each time there is a change to this first node, the object must send a signal message to the second node. Depending on where the second
node is located, this communication can take place either within the same object or via network communication to another object.

5.1.3. Virtual Pocket
The third user interface method is called a virtual pocket. The virtual pocket takes its name from its real-world equivalent, which lets people store all kinds of useful things and carry them to other places. Virtual pockets are commonly used in computer games to establish interaction with the game world. The Reality Editor uses the virtual pocket to provide the user with several useful interactions. For example, the virtual pocket allows users to access a tool called the memory bar, which keeps screenshots of objects, so that their augmented reality user interfaces can become remote-controlled. The virtual pocket stores general-purpose user interfaces that can be dropped onto connected objects. These general-purpose user interfaces allow the control of connected objects or allow the Reality Editor to read the states of nodes. Additionally, the virtual pocket allows a user to add virtual logic nodes next to the earlier described nodes. These logic nodes provide a visual programming interface (later described as logic crafting) for the user, so that signals coming from object nodes can be logically processed before they continue to another node. Once a user has programmed a logic node, the virtual pocket can be used to store these logic nodes for further applications on other objects. As such, the virtual pocket is a powerful tool to enrich connected objects with all kinds of useful content.

5.1.4. Settings
The settings button is not a user interaction mode but, as the name suggests, a button to enter the settings. The Reality Editor introduces a variety of functionalities that can be enabled via the settings menu. There is a setting for extended tracking, which allows the marker to be tracked even if it is off-screen. This is helpful when the user interface is bigger than the visual marker itself. There is an object-locking setting that allows a user to lock nodes and links. This setting is enabled with the fingerprint scanner on iOS devices. The instant connection setting allows a connectivity method enabled by touching “nodes” directly on objects with the user’s hand instead of pointing the Reality Editor directly at the object. “Found objects” displays all objects in a local network, which is useful for understanding what objects are visible to the Reality Editor. The developer menu provides settings for AR/UI repositioning, which is explained in
more detail in the Developer Tools chapter. A clear sky mode renders all user interface buttons invisible. This can be useful for video recordings or presentations, where the buttons should not be visible. The reality UI setting allows the editor to show alternative buttons that report their functionality to the augmented web user interfaces. This could, for example, enable a search mechanism in a grocery store setting. Chapter 7.12. Visual Search will explain the search mechanism further.

Another useful functionality in the developer menu is the “external interface URL.” Since the entire Reality Editor user interface is web based, the interface itself can be loaded from a remote location. This remotely loaded user interface is then able to interact with local objects. This has proven to be an extremely useful method for programming the Reality Editor, since the code compilation boils down to a simple refresh of the editor webpage.

5.1.5. Freeze

The last button in the Reality Editor user interface is the freeze button. The freeze button allows a user to instantly freeze the video feed. Every connected object that is visible on screen stays interactive. On release, the Reality Editor continues using the video feed for interaction.

5.2. Memory Bar and Visual Pointer

The memory bar allows a user to save a still image of a connected object for later recovery. The image is saved one of maximum five saved images. Limiting the number of saved images keeps the memory bar clear, simple to use, and quick to access. It also prevents the Reality Editor from becoming a cluttered remote control, since the focus of the Reality Editor is the special interaction it allows between connected objects. Once a user clicks on a saved image, the image expands to full screen and the connected object becomes interactive again. Additionally, this limitation prevents the user from being overwhelmed with unlimited galleries of pictures. A user should be able to memorize just the last items in the memory bar and continue interacting with the world via the Reality Editor video feed.
5.2.1. Memorizing

Figure 28 An illustration shows the steps a user undertakes to save an object reference to the memory bar. a) The user taps on the screen. b) The Reality Editor exposes a big virtual pocket. c) The user drags the finger onto the virtual pocket. Instantly a miniaturized screenshot of the video stream is attached to the finger. At the same time, the memory bar becomes visible. d) The user drags the miniaturized screenshot on to the memory bar to save it.

To save a still image of an object (Figure 28), the user points the Reality Editor at a connected object. Once the user interface comes up, the user taps the screen outside of the connected object user interface. The menu buttons change into one big virtual pocket. Once the user moves his or her finger on the screen onto the virtual pocket, the memory bar appears on the top of the screen. (While the illustrations show seven slots in the memory bar, the working prototype showed that seven images next to each other are too small to be useful. The actual Reality Editor application, therefore, has only five slots.) At the same time, an image recording of the screen sticks to the finger. Once the user drags that image onto the memory bar, the image is saved in the memory bar for later access.
5.2.2. Retrieving

Figure 29 An illustration shows the steps a user requires to retrieve an object reference to the memory bar. a) The user taps on to the virtual pocket button. b) The previous step opened the virtual pocket and exposed the memory bar. The user taps a saved memory. c) The Reality Editor shows now a still image of the saved memory. All interfaces associated with objects shown in the memory are fully functional.

If a user wants to retrieve an item from the memory bar (Figure 29), he or she can tap the virtual pocket. The virtual pocket screen opens up and shows all of the saved items in the top memory bar. A user activates a memorized item, or memory, by simply tapping on it. The virtual pocket menu disappears and the saved image of the item becomes visible. From this moment on, the image acts as if the freeze button were activated. Pushing the freeze button will make the memory disappear (Figure 30).
Figure 30 An illustration shows how a user can return from a frozen image back to the normal Reality Editor interface by tapping the activated freeze button.

5.2.3. Deleting

Figure 31 An illustration shows the steps a user requires to delete an object reference from the memory bar. a) The user taps on the virtual pocket to expose the memory bar. b) The user taps and holds a memory. After a moment, the memory becomes moveable and a trashcan button is exposed. c) Moving the memory onto the trashcan deletes the memory.
Deleting an item from the memory bar is similar (Figure 31). The user opens the virtual pocket interface via the virtual pocket button and taps on the memory to be deleted. Instead of instantly releasing, the user holds the finger in place until the button menu changes into a big trash can icon. At the same moment, the memory becomes movable again. Dragging the memory onto the trash can deletes it. However, since placing a finger on a memory makes it moveable, the memory can also be moved to a different location.

5.2.4. Visual Pointer

![Visual Pointer Diagram]

Figure 32 An illustration showing how a user can interact with visual pointers. a) The Reality Editor shows a memory, indicated by the active freeze button. Tapping on a visual pointer opens another frozen memory (b). c) The Reality Editor shows two objects in a video feed. The user drags a patch wire from one object node to a node on the second object. d) Once the second object is taken out of the camera feed, the patch wire instantly flips to a visual pointer representation.

An image of an object stored in the memory bar is saved in the connected object itself. Therefore, others can access memories without recording them into their own Reality Editor.
instances. A memory is a visual data pointer, pointing to a remote memory address. When two objects are connected (e.g. a light switch on the wall and a light on the ceiling), they are related to each other and this relationship should be represented in an augmented reality user interface, even if one object is not physically present. In this case, the Reality Editor introduces visual pointers. A visual pointer is an image reference that floats around a visual node connected to a remote object. For example, if object A has a connection to object B, but object B is not visible, then a visual pointer floats around the connected node of object A. The visual pointer behaves much like a memorized item stored in the memory bar.

Several interactions are possible with visual pointers and connections between nodes.

1. A user can tap on a visual pointer to load the memory of the object that is associated with the pointer (Figure 32/a-b).
2. A user can drag a connection from one node associated with object A to another node associated with object B in order to generate a connection. Once object B becomes invisible, the visual pointer becomes visible, pointing to object B (Figure 32/c-d).
3. A user can drag a line from a node associated with object A onto the virtual pocket. The virtual pocket exposes the memory bar, onto which the line from the node associated with object A can be dragged. Once a memory of a stored object C is recalled, the user can drop the line onto a node of object C to establish a connection between a local visible object and a remote object. The connection becomes instantly visible with a visual pointer attached (Figure 33).
4. If the connection to a remote object is deleted, the visual pointer disappears as well (Figure 34/1-2).
5. If the user taps a finger on a visual pointer, the bigger virtual pocket icon becomes visible again, indicating that the pointer can be stored in the memory bar. Dragging the pointer onto the virtual pocket makes the memory bar visible again. Dragging the visual pointer onto the memory bar saves the visual pointer onto the memory bar where it stays, even if the user removes connections associated with it (Figure 34/2-3).
Figure 33 An illustration of how a user can connect a visible object with a previously saved object.

Figure 34 An illustration of how a user can remove a visual pointer (1-2) and save a pointer in the memory bar (2-3).
5.3. Object Locking

The Reality Editor allows a user to become a master user of objects. This means that the user can monopolize access rights to an object as long as another user has not already prevented such access. Once a system is visually connected and all of the settings for the objects in it have been configured, a master user can use a visual locking tool to allow reliable access control for connected objects.

For this purpose, the Reality Editor includes a setting that allows any user to become the master user, by locking connections and objects. When the user opens the settings menu, types a secret passcode into the object locking menu, and pushes the activation button, the locking tool becomes visible. The Reality Editor confirms the user’s identity with the iPhone’s fingerprint reader, at which point the secret passcode becomes invisible and all future locking activities are saved with the passcode. For another user to gain access to the same locks, the second user’s Reality Editor must be set to the same passcode.

Locking and unlocking items in the Reality Editor is simple. Pushing the locking button locks every connection, node or object visible onscreen, and pushing the unlock button unlocks them. This locking mechanism is useful, for example, when a parent wants to configure a family home and prevent the kids from changing the set configurations. Such scenarios could include a door-locking setup, entertainment system controls, or a garage door opener safely locked to a desired functionality. Only the secret passcode will allow a user to reconfigure the devices.
5.4. Developer Tools

5.4.1. Reality Editor UI

Who are the designers and engineers who will develop for the Reality Editor? This was a question I was asked in a meeting with designers from IDEO in 2013. My ideal is to lower the barrier of entry for the creative disciplines. This means that the core creative process should be as flexible and as simple as possible. The underlying web technology implemented in the Reality Editor provides some of this flexibility. Because the Reality Editor supports standard web content built with HTML5, CSS3 and JavaScript, every web developer tool in the world can be used to create content for the Reality Editor (Figure 35). No proprietary editing environment is necessary. The content is still two dimensional and needs to be positioned in 3D space relative to the trackable marker, which would normally require knowledge of 3D software or an understanding of 3D graphics programming. However, the Reality Editor provides simple, built-in visual developer tools that enable web designers and developers to build augmented reality experiences without any specialized knowledge.
Once the developer mode is activated from the developer settings menu, a stripe pattern overlays the user interfaces augmented onto trackable markers. This stripe pattern (Figure 36) indicates that the object can be repositioned in 3D space, and that the user interface is not currently responsive to user interactions.

![Figure 36 A screenshot from the Reality Editor that shows a user interface that is currently moveable, as indicated by the visible stripe pattern.](image)

Two interaction methods are enabled in this mode:

1. Placing one finger on the user interface allows the developer to move the user interface on the x and y axes parallel to the trackable marker (Figure 37).
2. Placing a second finger on the screen allows the user to not only move the user interface but also scale it. Pinching out increases the size of the user interface, while pinching in decreases the size (Figure 38).

When the developer mode is activated, two new buttons appear in the user interface. One is a reset button that allows the developer to reset the interfaces back to their original origin positions. The second button lets the developer reposition the user interface in 3D space and make use of the freely movable position of the smartphone running the Reality Editor. Since the smartphone position relative to the trackable marker is known, the user interface can be locked...
relative to the smartphone by tapping on the user interface. When the developer taps and holds on the user interface, its position in 3D space relative to the smartphone locks, so that when the developer moves the phone in space, the user interface position moves with the smartphone as well. Once released, the user interface is placed at the new position. This interaction feels much like using tweezers to reposition objects that cannot be touched with the hand.

*Figure 37* A developer uses the Reality Editor developer tools position a web interface

*Figure 38* A developer uses the Reality Editor developer tools to scale a web interface
5.4.2. Reality Editor Server

The Reality Editor works with standard web technology, such as providing content via the HTTP protocol. The Reality Editor server build as a web service that provides in a local network all the capabilities connected objects require. Most of these capabilities are automated in the background, and are well explained in the System Implementation chapter. Some front-facing user interfaces, however, support the fast development of user interfaces, and are accessible via a web browser interface (Figure 39).

![Hybrid Object - Administration](image)

*Figure 39 Screenshot from the Reality Editor Server developer web interface.*

The Reality Editor server can host a multitude of user interfaces, which means multiple objects with multiple tracking markers. These combinations of user interface, object, and tracking marker are called hybrid objects or connected objects. To create a new object, a developer only needs to click on the “create new hybrid object” button in the Reality Editor server user interface.

Each object contains several selection buttons:

1. The info button provides a fast overview of all of the relevant connected object’s information. For example, the info menu shows all links and a live feed of all nodes
associated with the object. Furthermore, this menu shows the object ID and what other objects are known to the object.

2. The add target button allows the developer to enter a menu specifically designed for generating trackable markers associated with the connected object.

3. The add interface button opens a menu that shows the folder structure of the object-related static web server. HTML files stored in this folder structure will be loaded onto the connected object and trackable marker as user interfaces. For convenience, a developer can simply drag and drop files onto this screen to store them in the object folder.

4. The backup button allows the developer to download the entire object, with its user interface, trackable marker, and the 3D positions of nodes and user interfaces. The object can be restored by either opening a selection menu with the upload button or again just dragging and dropping the saved .zip file onto the browser window.

5. The delete button deletes an object.

5.4.3. Implementing a New Connected Object

There are three options for implementing a new connected object.

1. A disk image for an Arduino Yun can be cloned onto an SD card, which is placed into the Arduino Yun’s SD card reader. As soon as the Arduino Yun is powered up and the operating system is loaded, the Reality Editor server starts running. From here, a Reality Editor Arduino library can be used to program new connected objects with the API described in Chapter 12.3. and the developer tools can be used to build new connected object user interfaces.

2. The Reality Editor server can be executed within the Windows, Linux or OS X operating system. As a baseline, wherever the Node.js runtime environment works, the Reality Editor server will too. The Reality Editor server comes with a plugin structure that is stored in the hardware interfaces folder. Each hardware interface stored in this folder can communicate to a specific peripheral device. For example, one preprogrammed hardware interface allows the Reality Editor server to connect to the Philips Hue lighting system, while another allows it to connect with the general inputs and outputs of a Raspberry Pi board. No matter what hardware is used, a hardware interface needs to be implemented to
talk to it. The API used for implementing hardware interfaces can be seen in Chapter 12.2. If a hardware interface for the desired hardware already exists, the developer can simply enable it via the enable variable, and may also include some credentials in the code.

3. The last method is similar to the second method. Because the server is built with Node.js, there are many libraries that allow developers to program and connect with existing smart devices. A hardware interface can be used as a bridge between the Reality Editor server and the web service of a smart device.

Each of these methods requires programming skills for the initial setup. However, Chapter 6.10 presents a stand-alone application that instantly connects to Lego WeDo 2.0 sets and the Philips Hue lighting system. This application uses the server with a customized hardware interface implementation that provides a user-configurable visual frontend. A future goal is to provide more such stand-alone configurations or build the Reality Editor server right into connected objects.

5.5. Touch Connections

The Reality Editor infrastructure at its core allows decentralized connections among nodes of connected objects. The Reality Editor enables a user to connect nodes visually with augmented reality. However, once such infrastructure is in place other methods can be used to create similar results. For example, a user can establish a connection among two objects by touching the origin object and then touch the destination object. If both objects have capacitive touch sensors, the touch can be used to facilitate a connection.

This is useful, where a visual identifier is not beneficial. Another method for identifying the user interface components and its nodes is required. For example, the brightness of a light and a separate light switch could be identified with a capacitive touch sensor. Both devices are prepared to send special action messages that indicate a touch into the network. When a user touches the light switch an action message is sent to the system that the origin for a link is
activated. When the destination light is touched, the link is complete and will be saved in the origin object. The Reality Editor is used as a negotiator between these two devices to create this new touch connection. This means that this functionality only works if the Reality Editor application is open and the instant connection functionality is enabled. This helps to prevent error entries when a user is actually operating the light switch, because touch connections can be created or deleted only if the Reality Editor is open.

The Reality Editor also provides the possibility to mix touch connections with the regular augmented reality user interfaces (Figure 40). As such, a user can touch an object to bring up the touch connection user interface in the Reality Editor. Instead of touching another object, the user draws a line from the instant connection user interface onto a visual node of a connected object. This also works the other way around, depending on how the control flow should appear.

The instant connection method shows the potential for the Reality Editor infrastructure to be used in different ways. It also shows an interesting semantic, where namespaces are defined by objects and functionality nodes within the objects. This semantic could enable voice interactions with objects, where the Reality Editor infrastructure allows flexible voice control of objects.

*Figure 40 Illustrations that show the visual feedback on the Reality Editor while an instant connection is created.*
Touch and augmented reality may allow a user to focus a voice command onto a specific object in one room. For example, a user can touch or point a mobile phone at a light while saying that “the brightness of this light” should be controlled by “this switch” (where the user has moved the mobile phone to point at a switch on the wall). In this example, the user’s position relative to an object helps to specify the destination of the voice commands, while the Reality Editor system helps to make sense of the voice command.

5.6. Logic Crafting

Figure 41 The Reality Editor is pointed at the armrest of a chair. The armrest exposes data points that represent the chair’s functionality.

The development of a multitude of use cases during Reality Editor workshops exposed the limitations of the simple connections between components previously described. Often the user wants to change the behavior of the connection itself and thereby how one object interacts with another. For example, some users wanted to specify that when a sensor in an office chair detects that the user is leaving, the desk light should be turned off after five minutes. The simple connections discussed so far can represent an IF THIS THEN THAT rule. However, often we want to be able to specify IF THIS THEN HOW THAT types of rules. We initially
circumvented this problem by mapping logical functionality to a multitude of additional functional nodes. This solution worked but was not very elegant and did not match our overall vision of separating a thing into its functional components. For example, the chair would show a multitude of data points for sensing whether a user is sitting in it (Figure 41) instead of exposing just one data point representing the pure component. I concluded that a general programming interface was needed in the Reality Editor—that is, a general tool to define HOW one data point relates to others. The question then became, what should such an interface look like?

From an engineering perspective the best solution would be to provide a high-level programming language. Such a programming language would allow a programmer to read incoming data samples from another connected object and initiate tasks based on defined logic. This model provides a great amount of freedom, but it also has a number of disadvantages.

1. It introduces another layer of user interface that follows different methods than those previously proposed in this thesis and is much less accessible to a wider audience.
2. It is based on the desktop paradigm and as such it is disconnected from directly mapped user interfaces.
3. Functions in programming often follow a control flow paradigm. This means that a program is written in such a way that it executes one task after another. Interacting with life data streams in such a way requires the programmer to create repetitive sequences in the code that analyze the life data stream step by step and eventually call another task based on certain result triggers.
4. One needs to know all the possible commands, function calls and classes to write working code.

A primary goal for this thesis is to provide intuitive, self-explaining user interfaces that a user can operate right away, without needing to be an expert. This is the main argument against the use of text-based programming as an additional user interface for the Reality Editor. The interface that is used for programming connected objects has to be visual and self-explaining.

There is a multitude of visual programming languages (VPL) out there [80]. Two directions are particular interesting. For one there is visual data flow programming, which is the most used
visual programming direction [80]. Visual programming allows a user to see how data flows through a program instead of organizing the programs with functions that are iterated with a control flow. The second interesting direction is the block-based programming [81], that provides visual guidance to the user by representing each programming functionality with a visual block. A user can choose from a set of options and arrange visual blocks in such a way that a programming structure appears, similar to what a text-based programming language would provide.

The visual user interfaces for data flow systems can look very different from block-based programming or text based programming. For example, when a user builds a logical loop to iterate through a sensor reading function in order to generate a real-time response, the data flow system is built to act on each individual sensor reading. Each sensor data unit becomes a signal that the system acts on.

From a desktop programming perspective, a data flow system might seem cumbersome to operate, but from a physical objects perspective, data flow has a lot of advantages:

1. Since the system treats all signals as real-time data, a user does not need to consider the difference between a stream of signals or an individual control action.
2. A user can generate actions and reactions with a simple connection from one node to another. In a control flow system, a user would need to generate functionality that reads signals in intervals and then executes an action when needed.
3. Data flow systems are simpler to visualize, since data is flowing from node to node. In a control flow system, the data is separated from the visual program. As such the visual representation becomes more complicated, or the overall understanding of the system becomes more difficult.

On the other hand, data-flow systems have certain disadvantages when it comes to the generation of complex programs. A real-time data flow system makes it more difficult to trigger subsystems that perform independent tasks or tasks that require a different processing time. It also makes it very difficult to build systems that repeat tasks based on a single signal. The overall control for individual time-based processes are difficult to model.
How can a visual interface provide a simple direct method for connecting physical objects with logic mapped into the physical space? How can a system provide a data flow-like user interface that at the same time allows for more complex programs that can follow control flow?

![Diagram](image)

**Figure 42** Virtual data points are connected with the data points owned by objects. Each of the robot’s distance sensors is connected to a virtual scaling node connected to a motor. The same distance sensors are also connected to an inverter node that is connected to a motor on the opposite side. The scaling nodes let the robot drive forward. However, when it hits a wall, the inverter sensor becomes dominant and slows down the opposite motor. As a consequence, the robot moves away from the wall. This demo is possible because it processed the state of the entire robot at once.

Different approaches can be pursued to solve this programming interface problem. One approach would be the creation of virtual data points. Virtual data points are data points that a user can place next to the data points representing the object components in 3D space. The user can then connect a data point owned by the origin object to the virtual data point (e.g. a 5-minute delay), and that virtual data point can be connected further to the data point belonging to the destination object (Figure 42). The virtual data point changes the transmitted data according to its program.

Such virtual data points were included in earlier versions of the Reality Editor. Implemented virtual data points allowed inverting, triggering, delaying, and scaling of data. These virtual data
points worked well to change the mapping of data that would flow from one data point to another. However, their functionality turned out to be too limited when defining more complex interactions. For example, a user should be able to model a simple idea such as: “If it is night and the apartment door is opened, switch all lights in the home on for a defined period of time.” A corresponding rule would look like: “If two sensors (brightness sensor for the sun and a distance sensor for the door) report a reading smaller than some defined thresholds, a switching element is set to one. A delay element will delay the switching signal and send its delayed signals to the same switch to activate the switching back to zero. The connected lights will turn on with the first switching signal and off with the second.” With the first generation Reality Editor, it was very hard to model such a simple behavior.

A second approach to the problem is to adapt methods from data flow programming languages such as PureData [82], as they appear similar to the Reality Editor in their use of data points. The problem with this approach is that while this type of programming is useable by an expert, it becomes too complex for non-experts. Virtual objects with virtual connection lines quickly overwhelm the AR interface. A more appropriate method had to be found that enables a user to create simple logical stories for their connected objects.

Figure 43 A screenshot of The Secret of Monkey Island showing the game world, a command panel, and the player's inventory. The user can store found items in this inventory.
A promising inspiration for such a tool can be found in the combination interfaces of computer games. 1990s adventure games such as *The Secret of Monkey Island* (Figure 43), *Maniac Mansion* and *Day of the Tentacle* introduced the concept of a player's inventory menu that allows the user to collect and store items to take to different locations in the game. In order to advance in the game, the player had to use these collected items in other locations. Items could also be combined to form new items that could be stored in the player's inventory or used in the game world. At the time, these adventure games were highly successful, and the player's inventory was an easily understood and functional metaphor. Since then, many other games have adopted this kind of user interface. The most successful is the game of *Minecraft*, which models a world made out of blocks. Each single block can be destroyed, collected as a resource or placed into the world. A user can combine blocks in many different patterns and variations to craft new blocks and items that then can be placed back into the game world. Such crafted blocks and items can themselves be a resource that is used to craft other blocks and items of a higher order. Items are combined using a metaphorical “crafting table” (Figure 44), which is limited to a 2x2 or 3x3 grid of options. The limited space on the crafting table together with an enormous diversity of blocks supports almost unlimited creativity in crafting items and applying them in the game world. The simplicity of the crafting table in combination with the multitude of blocks results in a simple workflow with a rapid learning curve. For example, once the mechanism is understood, a user can look up on the Internet how to craft different items. In the case of Figure 15, three sticks combined with thread creates a bow. The sticks themselves are crafted from three pieces of wooden blocks, and the thread can be found in temples located in the game world. In the case of
Minecraft, the metaphors of the player's inventory and the crafting table result in a tool that enables a huge number of possible combinations, yet maintains high usability.

This opens up some interesting possibilities for the Reality Editor. For one, a player's inventory can help to structure and organize programming nodes. Second, a system that allows almost unlimited possible combination of blocks with a simple usability, can help to form a visual programming tool that allows a better experience that does not overcrowd the visual space. Such a different visual representation can bring more structure into the data flow program, so that a user better understands the flow of the created program. I therefore created an interface concept, called the logic node, that allows a user to place a special node within the AR interface onto a connected object. When the user taps on such a logic node, a new static interface pops up that does not use the augmented reality view.

![Logic Crafting Interface](image)

*Figure 45 This logic crafting interface appears when the user taps on a logic node.*
Figure 46 When the user taps on the green virtual pocket button, a block selection menu appears.

Figure 47 When the user places a finger on a block and starts moving it, the block attaches to the finger and the block selection menu disappears so that the user can place the block on the logic crafting board. The block is red when it can not be placed.
Figure 48 If the user places the block on a free grid position, the block becomes green. This color indicates that the block can be placed at this position.

Figure 49 A logic block with a size of three blocks is placed onto the logic crafting board.

Unlike interfaces that are directly mapped into an augmented reality video stream, this new interface (Figure 45) consists of a static screen organized into a grid of 4x4 block placeholders. A virtual pocket, inspired by the earlier described player's inventory, holds a multitude of logical elements, each represented as a small block (Figure 46). A user can place these blocks onto the 4x4 block placeholder grid by tapping and moving a block. When a user starts moving a block within this block selection, the block selection disappears, so that the user continues to move the block in the 4x4 grid (Figure 47) which allows the user to place the block onto the grid. Once the block is released onto the grid, it sticks on top of one of the 16 placeholders. When a block has one input and one output, it is only the size of one placeholder block. However, if a block has
more than one input or output, the block can span the size of multiple placeholder blocks (Figure 49). For example, two input or outputs would make a block that spans two placeholder blocks.

There is enough space between these placeholders so that a user can draw lines among placed blocks. The blocks in the topmost row are connected to the inputs that can be dragged onto the logic node in the augmented reality interface. The blocks in the bottom row are connected to the outputs of the logic node. Each column is color-coded, which makes four different colors. This color coding corresponds with the logic node in such a way that lines drawn from another node can be placed on each of the four colors, and a color can also be chosen when a line is drawn as an output from the logic node to another node. As such, the color coding allows the system to connect the “inside” of a logic node to the “outside” of the node. This kind of organization allows for a visual interface where data flows from the top to the bottom, and is named logic crafting. The 4x4 grid is called the crafting board.

This data flow interface looks much like control flow oriented programming environments, where a program is usually structured from top to bottom and from left to right. Unlike other data flow interfaces, where the structure of the flow tends to be left entirely to the user, the described interface creates a solution in between, where the structure flows from top to bottom and the user has some degree of freedom in placing the blocks in the programming interface.

Other systems show similar overlaps but from a control flow perspective. Scratch [81], for example, uses visual blocks for programming that resample control flow programming. In some cases where real-time data is processed, however, the interface uses data to control input and output. Repeating control flow mechanisms are automated in the background to make the system more intuitive. It seems that when a program is dealing with real-time data input and output, a data flow system is more intuitive. It can easily be described like the flow of water or the flow of electricity. However, a data flow system requires program units that break the flow of real-time data down into individual tasks act on the referenced data. A system of program loops is usually used to represent real-time data in control flow. An analogy could be the logic of a sluice system, whereby the data flow is describing the flow of water itself.
The difference between the data flow and control flow lies in the specific referencing of data. Control flow mechanisms tend to focus on the execution of tasks executed one after another. Data processed in these tasks is referenced by names that point to a fixed data storage. As a consequence, all tasks need to wait for each other when executing on the same data. Data flow, by contrast, does not separate the data from the flow of the program. A data package is sent to a program unit and executed. The resulting data is sent to the next node. Data flow does not require a reference to data storage, because the program sends the data from program unit to program unit. This means that a program unit can execute on an entirely unique data package; at the moment the data package is sent to the next unit, new data can be processed. This is beneficial for real-time systems such as the Reality Editor. Scaled to a real-time system of connected computers or connected objects, the data flow allows a true decentralized system, because at no point do the individual units need to reference a central database. As the units do not need to wait for central data for synchronization, the system is scalable.

Another important point is that when the data is sent from computer to computer, each computer can provide its own method for processing this data. This means that each computer only needs to know the type of data to process. Control flow systems are weak in this regard, because they require the encoder (origin computer) and decoder (destination computer) of the control command to perfectly match one another. As an added component of complexity, the referenced data and its state need to be as expected. Therefore, a control flow system would require all of its components to match perfectly. That is not possible in a decentralized system, where all of the components can be at different stages of development, with different version numbers or modifications.

All of these considerations argue for the data flow user interface and decentralized networked programming system implemented in the Reality Editor. However, there is a specific situation in which a control flow system is more powerful. This is when a real-time system should act on the edge of a real-time data input instead of on the real-time data itself.

Because the edge of a real-time data input is basically a control flow input that occurs one instance at a time, the processing of this single input requires a sequence of program units that
behave like control flow programs. These program units can even act like full data flow program units, when referencing external networked data storage. For example, a switch could initiate the request for a door lock to ask a database if a user can get access. Simpler programs, could show a sequence of loops and delays that generate a choreography of actuators. These (literally) edge cases require specific handling. Let’s start with an example: An IF block can be the size of two placeholder blocks and route an incoming signal to the left or right of the block output, depending on a second statement input. The statement input can be a threshold (edge) that decides based on a second input signal. For triggering a start condition, the IF statement can also act as a switch that sends out a 1 or 0 on the left or right side, depending on when the statement flips. A delay can delay a signal by a defined time and a trigger switch can flip between 1 and 0 for each value reaching bigger than 0.5. These three elements can be placed on the crafting board, so that the block is placed on the top row so that it connects to the “outside” inputs of the logic node, the trigger switch is placed in the next row, and the delay in the third row. Connecting the left side of the IF statement to the delay and the delay back to the IF statement generates a loop, which can also be visually identified as such. Connecting the left side of the IF block to the trigger switch and the trigger switch to an output placeholder block in the bottom row will generate a switching signal that turns on and off depending on the delay timing. The loop will stop when the IF statement becomes lower than the set threshold and therefore the signal routed through the IF block is not passed anymore.

The visual understanding of such a loop structure appears visually similar to that of a loop in a control flow programming system. It would also be possible to build a loop block as a single block. The loop block executes a pre-defined number of loops when triggered on an input. But this shows the problem described earlier—when the edges appear more often than the looping system can finish executing them, the system breaks, because the data flow system does not wait for a task to end. The problem in a real-time data flow system is that a stream of signals can cause an unmanageable number of loops and program units that are executed in parallel, flooding the system and effectually destroying its real-time nature.

In case of the first loop, this is not a major problem, because the looping structure is separated from the incoming data signal. Once the left side of the IF is triggered, the loop will execute until
the signal stream becomes lower than the threshold. However, if a signal stream is injected into
the looping structure, the described flooding problem appears. In the second case, only a single 1
signal can be sent into the loop block for triggering the loop system. For as long as the loop is
active, a second signal would only generate a parallel executed second loop. This means that the
logic crafting interface can provide similar visual understanding of programs as can a block-
based control flow programming system, but it comes with an individual set of specific technical
problems.

One solution for handling control flow-like data flow program units is that there are two kinds of
signals that propagate through the Reality Editor system. One is a signal stream and the second is
an individual signal (edge). For example, an individual signal (edge) is generated when a user
pushes a physical button. The button will trigger a one when the button is pushed down and a
zero when the button is released. For the first described loop, this could mean that as long as the
button is being pressed, the trigger switch will execute. For the second loop this would mean that
once the button is pushed, a defined number of loops will execute. Most of the user interfaces a
user executes follow the button model. A stream of signals is triggered, for example, when a user
rotates a knob or moves a slider. In such cases the first loop will work, but the second loop
would fail.

A particular stress moment for the Reality Editor server exists when a signal stream is routed into
a loop that expects a single signal at a time. There are some solutions for these stress moments in
the logic crafting system. For one, the system can test its own real-time execution capabilities.
Once there are too many signals executed in the same time or the system requires too much
processing time, the system will slow down the execution of signals, to keep the system stable
and drop signal execution for when the buffer of signals becomes too big or the time delay of
execution is too far away from the required time. However, such a system is less optimal for
scenarios where every signal is important. One example for such a scenario could be machine or
robotic controls.

Instead of implementing the potential loss of data, the visual data flow system within the Reality
Editor crafting can provide additional feedback. For example, if too many signals are executed,
the connecting lines among blocks or nodes can increase their size and change their color. This can help a user debug the system. This would help for example with the first loop structure, where a signal stream is injected into the loop.

When looking at the second loop, the solution might be to allow only one instance of the loop to be executed. This would mean that if a stream of signals is arriving at the loop block, the first signal will execute the loop and the rest of the signals will be ignored until the loop is finished. Both loop structures have their advantages and disadvantages. From the perspective of programming blocks used in the logic crafting interface, it is important to understand the specific possibilities of system overload. Blocks for logic crafting that act on individual signals need to be able to handle streams of signals as well. But as such, the logic crafting interface presents an in-between stage of a very simplified control flow and data flow system combined.

As described earlier, the data flow system is particular intuitive when handling real-time inputs and outputs. Additional, the Reality Editor logic crafting interface provides an incremental learning curve, where a low barrier to entry allows a user to understand the basics fast and then gradually master more and more functionality. The Reality Editor accomplishes this by allowing a user to start out by drawing a simple line from one node to another. They can graduate to creating more complex programs by using the logic crafting interface, where the same method is used to connect one block to another block. It appears as if the logic crafting interface is a different representation of the same node-connecting method. However, virtual blocks can be placed on the logic crafting board and these blocks can be connected. From a mental model perspective, a user always just connects nodes, each of which represents data that flows through the system. The foundational concept for programming follows the same method as the simple connection of two nodes.

Once a user understands logic crafting, more difficult programs can be explored. A user can create certain control flows in the logic crafting interface. However, once a user reaches an expert level, where the full advantages of a programming language are wanted, the logic crafting interface is not the right tool. In that case, one may want to build individual blocks using a programming language and use the logic crafting interface for combining these new creations.
The Reality Editor system provides a simple JavaScript API for generating such blocks, the details of which will be explained in the System Implementation chapter. The learning curve for using logic crafting is as follow:

1. Learn to connect one node to another node.
2. Learn how to use the crafting board and how to add and connect blocks.
3. Connect with friends and online communities to learn about the best logic block combinations.
4. If the blocks are too limited for you, create your own blocks with JavaScript and contribute these blocks back to the community.

It is important to draw a clear line between what logic crafting is and what it is not. It is a tool that allows a user to place small logical ideas into the physical world. It is not a complex programming tool that allows users to model complex ideas. Complex ideas should be programmed in JavaScript and made available as logic blocks. For example, the Reality Editor can help to define the timing of a machine or execute a couple of delayed actions when a switch is pushed. However, a complex machine learning application should be programmed in JavaScript. The learning input can come from the logic crafting block input and a result for the calculations can be returned on the logic crafting block output.

Because the focus of logic crafting is very much on creating a simple tool for everyone, the complexity of each logic node program is intentionally set to 16 blocks. For one, this allows the users to keep an overview of the entire program, but it also allows for easy communication and sharing. A user can reproduce a program by simply looking at a logic node diagram. However, it also requires the block programmer to build blocks that are high-level enough. As a result, a user can choose from blocks that are more task-oriented, rather than requiring an understanding of programming. This is the goal for logic crafting.

The following are screenshots taken from the logic crafting board to illustrate some programs.

*Delay switch*
If a switch is the input and therefore only single signals are sent with each user action, the above program (Figure 50) can be used to create a delay switch. The program does the following: A threshold block is set such that only on signals are routed through. Off signals are ignored. The threshold block routes the signal to the output to switch the yellow output on. The same signal is also routed to the delay block. The delay block waits a specified time to forward the signal to an inverter block, which turns the incoming on signal into an off signal. The new off signal is then sent to the same yellow output and turns off the delayed output that was previously set to on.

**Delay cascade**

In the above program (Figure 51), the input signal is routed to all four outputs (blue, green, yellow, red). The first blue output equals the blue input in real-time. The signal for the second
green output, however, is routed through a delay block, so that the signal will exit the logic node via the green output delayed. This delayed signal is also routed into a second delay block and from there into a third. The signal is delayed more and more, each time that the signal propagates through a delay block. Eventually the red output receives a signal that has been three times delayed by a cascade of delay blocks.

*Seesaw switch*

![Seesaw switch](Figure 52 Screenshot from a seesaw switch logic crafting program.)

With this seesaw switch program (Figure 52), two inputs are routed into threshold blocks. The threshold blocks make sure that only a signal exceeding a certain value will be forwarded. This helps with fine-tuning the seesaw block. The seesaw block has three inputs and three outputs. The seesaw flips between sending an on or off signal on the first two outputs, depending on the signals coming into the first two inputs. If a signal bigger than 0.5 enters the first input and the first output was set off, then the first output is set on and the second output is set off. The same happens in reverse order with the second input. If the second output is set to on, then the third input will be routed to the third output. If the second output is set to off, then the routing between the third in and output is interrupted. The above program therefore routes the yellow input to the yellow output, if there was a strong enough signal on the green input.
In this example (Figure 53), two threshold blocks are set to either output on or off. The resulting signals are added. The resulting signal on the green output will stay on if one or both if the inputs are on and switch off in case none of the inputs are on.

**5.6.1. Design Considerations**

The logic crafting board introduced a couple of design problems:

1. How can a user combine simple programming blocks in the simplest way?
2. What visual supports can be built to support the user when combining blocks?
3. What is the best way to connect the outside augmented reality interface with the logic crafting board?
4. How can a minimum of user interaction methods be used throughout the entire Reality Editor? How can the entire system make use of the same simple concepts and gestures?
5. What structure will allow a user to learn how logic crafting works, step by step, yet will also be able to make meaningful interactions at any given point?
As described earlier, the idea behind logic crafting was not to build a comprehensive and complex programming language, but to enable a simple and fast tool for expressing ideas for how objects interact with each other. The logic crafting board emerged as a result of this idea, but other concepts were considered as well. Starting from the described virtual nodes, there was also an idea to take virtual nodes and place them on a board that would automatically snap inputs to outputs (Figure 54). This method of snapping blocks would allow for instant feedback and instant functionality. However, it would also make it difficult for a user to draw connections among programming blocks, meaning that it would have an easy learning curve but would not be extendable enough.

Another idea allowed users to just freely place virtual logic nodes onto a static space. This would resemble the look and feel of known data flow programming systems such as Pure Data [83]; however, it would give the user too many choices. It would mean that, as well as understanding how to connect nodes, a user would also need to know how to navigate in this new interface with zoom and pan. The space would set no limitations on the size of a program, and therefore the complexity of the programs could grow too great, shifting this tool into a professional niche instead of allowing to remain a tool for everyone to use. However, a benefit of such a tool would be that a user would be able to follow the same paradigms in the logic crafting space as are already used in the directly mapped augmented reality user interface. The described logic
crafting approach represents a well-balanced in-between solution. A user can drag and drop logic blocks (that act like nodes) onto a predefined grid. The logic blocks snap into position and provide guidance to the user. The grid only allows a maximum of 16 blocks to be placed, which guarantees that the complexity of each individual logic node stays clearly understandable. The grid also provides visual guidance to better understand a program, since the flow of data happens from the top to the bottom. As such loops are clearly identifiable. Similarly to the directly mapped user interface, a user combines blocks into a program by drawing lines from an origin to a destination block.

A variety of guidance systems have been implemented in the user interaction with the crafting board. First, when a user selects a block to be placed onto the logic crafting board, the block attaches to the finger and moves to the position wherever the finger is moved to. This allows a user to understand that the placement of a block is the present task. Once a user moves a block on the logic crafting board, another guidance systems supports the user in the form of a snapping mechanism that snaps logic blocks onto the next possible board position. A possible board position is a grid position that is unoccupied by another block and dedicated to holding a block. If a block is not snapping to a position or the closest position is already occupied by another block, the block that is attached to the finger shows red stripes. Once a block snaps to an open spot, the block shows green stripes (Figure 48). When a user releases the finger and therefore places the block on the grid, the block image changes from the described guidance system to the icon of the block functionality. If the finger is released while the block has a red color, the block disappears without being placed on the logic crafting board.

Each block in the grid has the same height. This means that all around the blocks there is space for connecting lines to show the flow. When a user draws a line from one block to another block, the connecting line automatically routes its way via these open spaces. When more than one line uses the same “channel,” the lines automatically position themselves to be beside each other. This system helps the logic crafting board to appear clear at all times. The lines will never overlap the blocks and the flow of data is organized. Similarly to the directly mapped user interface, a user is able to swipe through lines to delete connections.
A bigger design challenge was the connection from the inside to the outside of a logic node. “Inside” in this context means when the user taps on a logic node, and therefore opens the logic crafting board. The user metaphorically dives into the logic node to reprogram its functionality via the logic crafting board. “Outside” means that a user can see the logic nodes augmented onto connected objects as part of the Reality Editor user interface.

A user could, for example, move a connecting line onto a logic node. Once the logic node is touched, it could open up and the user could place his or her finger onto the right input. On the other side, a user could touch an output and after some time, the logic crafting board would disappear so that a connecting line could be placed onto a node in the directly mapped AR user interface. The problem with this solution, however, is that it is difficult to see the connections from the inside to the outside and vice versa once they have been set. Another idea is to expose a miniature version of the logic node inside. This would allow a user to connect a line exactly to the right spot on the inside. However, such a solution would still make it difficult to understand associations between the inside and the outside. Some kind of bridge needs to be generated that represents an exact spot on the outside in relation to an exact spot in the inside. This could be a written connector, such as “input 1,” which would be clearly marked on the input and the output. But such written connectors are difficult to grasp right away. Furthermore, it is difficult to provide simple understanding for multiple lines that would connect to one logic node. In testing different solutions, it became clear that color would be a simple tool to mediate such a bridge. On the outside, colors can indicate what node a line is connected to, and on the inside the same colors could help to understand the connections from the outside.

Figure 55 shows the different designs that were created to prototype how connecting lines could connect to a set of colors. Since the logic crafting board is organized in a grid of 4x4 blocks and each column has an input and an output, the logic node needs to connect four colors between the inside and the outside. The first version always showed a logic node with a circle of colors. If a user drew a line from the node, the output line would change its color to the color that had been crossed when moving the line away from the logic node. This idea worked well, but it was too crowded and too detailed. A second version was simpler and allowed for clearer visibility of the ring. Eventually, when placing a couple of these nodes onto a connected object, it became clear
that this idea would be distracting and take up too much space. Instead, the final version implements a logic node that shows the additional functionality of the ring only if it is required for the current interaction step. Logic nodes only differ in shape by being squared instead of circular. When a user taps on a logic node to draw a line to another node, the logic node expands into a four color selector. The same happens when a user draws a connecting line from another node to the logic node. Once the finger reaches a defined distance from a logic node, the logic node expands into the color selector so that a line can be dropped onto one of the four colors. This interaction is made possible with a hexahedron that spans around a logic node. The hexahedron is two-dimensional, but its size changes according to the z axis of the logic node transformation matrix. As such, a simple two-dimensional ray-casting algorithm [84] is used to check if the user’s finger position is inside the hexahedron or outside. Figure 55/4 shows the final appearance of the connecting color selector.

![Figure 55](image.png)

*Figure 55 Different designs for how a logic node appears when connected.*

The connecting lines themselves support this visual system as well. The system is prepared to also place input and output names onto each color. The functionality to place names on to the colors is explained later in this chapter. When a line is connected from a node to another node, it
is white. The line itself is assembled of dots that move from the origin to the destination and therefore indicate the direction in which the data is flowing. When the line is dropped onto one of the color selector colors, half of the line changes its color to the selected color on the color selector. When a user is not touching the screen, the logic node has no color selector, but the lines still remain in the selected colors. As such, the lines visually indicate to the user at any time how a line is connected to the inside of a logic node.

To further support the connection between the inside and the outside of a logic node, the IN and OUT blocks within the logic crafting board have an additional functionality. Whenever a user taps on a logic node to open the corresponding logic crafting board, the AR video stream is frozen. Once a user rests a finger on an IN or OUT block, the crafting board becomes transparent, unveiling the frozen video stream and revealing how the logic node is wired up. This functionality lets the user peek behind the curtain.

![Figure 56 Draw a line to connect to nodes.](image)

![Figure 57 Swipe through a line to disconnect the nodes.](image)
It took some time to combine the Reality Editor and the logic crafting user interface into a simple and consistent user interface experience. The basic interactions are: 1. draw a line to connect (Figure 56); 2. Swipe through a line to disconnect (Figure 57) and 3. hold and move to move nodes and user interfaces around (Figure 58). The logic crafting interface introduces a third interaction: Tap and move. Since this gesture is not particularly different than hold and move, one can say that the entire Reality Editor user interface can be operated with these three simple gestures.

To start the interaction with a new logic node, a user taps on the virtual pocket and drags a new logic node onto the directly mapped AR user interface of a connected object. Once tapped onto a logic node, the logic crafting board interface opens up. When a user taps on the virtual pocket button again, a selection of logic blocks is displayed. To support this change in the button function, the virtual pocket icon is slightly changed (Figure 60) and green in color. A user can drag a button onto the logic crafting board by simply tapping and moving the desired block. Instantly the block selection disappears and the block is movable on the logic crafting board. The
wiring of the logic blocks follows the method for wiring the AR nodes outside of the logic crafting board. Blocks become movable when they are hold and moved. The same method is possible with nodes and user interfaces in the AR interface.

![Image of a settings menu for a threshold logic block](image)

**Figure 60** A screenshot from the Reality Editor showing the settings for a threshold logic block

Setting menus can be customized with HTML and JavaScript by designers or developers who build logic blocks.

Each logic block has the option to provide a settings menu. When a user taps on the block without moving it, this setting menu can appear and allows the setup of the block. The settings menu for each block is webpage provided by the logic block plugin saved on the server. The setting menu is entirely defined by the block designer so that it can reflect the needs for each block configuration. For example, a threshold block can have a slider for setting the threshold sensitivity, a switch to define if lower or higher values should be cut off, and a switch that switches the output from digital or analog (Figure 60). The block is fully functional without the user entering the setting menu. However, if some changes should be applied, the user can dive deeper and apply these changes in this settings menu.
5.6.2. Handling Logic Nodes

Each logic node has a setup functionality that is accessible when clicking on the setting button within the logic node (when tapping on the logic node). Each logic node has settings for the outside appearance and the inside behavior. For the outside appearance, a user can set a logic node icon image, the name of a logic node and provide names for all 4 inputs and outputs. For the inside behavior, a user can define if a user will see the last opened logic block settings first or the logic crafting board when tapping on a logic node. This last definition is useful for when a user just wants to add a logic node to a connected object, without dealing with a program. This supports the simple learning curve of connect first, move onto logic crafting second, and eventually begin programming JavaScript (Figure 61).

![Figure 61 A screenshot from the settings menu for the logic node. The overall logic node setting allows a user to define names for each color input and output. It also allows the user to select an image as icon to be placed in the logic node. A switch can set the logic node to open the last block settings menu, for when a user taps a logic node. This allows a user to make changes to the behavior of the logic node, without being exposed to the logic crafting board.](image)

Once a logic node is programmed and its outside appearance is set, the user may hold his or her finger against it. When the logic node becomes moveable, the sidebar changes to a trashcan and a big virtual pocket button (Figure 62). A user has the choice to either remove the logic node by dragging it into the trashcan or save an instance of the it in the virtual pocket. When saving, the logic node is dragged onto the virtual pocket icon and then onto a free spot in the appearing virtual pocket register. Since the Reality Editor is a distributed system in which every user can
see all the programs, a user can program a logic node and another user can save this logic node into a personal virtual pocket. Logic nodes that are saved in a personal virtual pocket within the Reality Editor can be applied to any connected objects. For example, two children can play with a toy that supports the Reality Editor and program the toy together. At the end of the day, the visiting child can save the shared crafted programs to her Reality Editor virtual pocket to take them home. At home the child can drag and drop the saved programs onto her own toys and continue playing. When the children meet again, both can apply to a shared toy the programs that they have been creating at home.

![Image](image.png)

*Figure 62 Once the Logic Node becomes moveable, the sidebar changes its functionality to allow the user to either trash or save the node.*

### 5.7. Instant User Interface Elements

The Reality Editor allows the simple creation of directly mapped augmented reality user interfaces for bi-directional interaction with connected objects. These directly mapped interfaces are created with web technologies such as HTML, CSS and JavaScript. While researching new user interfaces for connected objects, a few user interface elements showed up over and over. Some of these elements were provided in examples to the Reality Editor community and used in workshops. Initially there was the idea that users, designers and engineers would use these general examples as a starting point to understand the Reality Editor API and build new user interfaces. However, over two years of exposure to the community, there was no significant
further development of these elements. Instead, many demos continued to use the general user elements that were initially provided. During some of the workshop discussions as well after feedback from the community, it became clear that the Reality Editor should provide a row of user interface elements. These elements can be dragged and dropped, similarly to how the logic nodes can be dragged onto an object.

These instant user interface elements were implemented into the Reality Editor as part of the virtual pocket interface. When the Reality Editor is set to programming or crafting mode, the virtual pocket exposes only functionalities related to the use of the logic node. However, when the individual interface mode is active, a tap on the virtual pocket exposes another selection. This alternative selection shows instantly selectable user interface elements (Figure 63). These elements drop onto connected object in the same way a logic node does. Each user interface element exposes a node in the crafting mode. This means that when switched to the crafting mode, a user can wire up the user interface elements with the connected object in order to make them responsive or interactive.

Figure 63 A screenshot from the Reality Editor showing selectable user interfaces. The interfaces are animated in the Reality Editor so that they reveal their capabilities to the user.
The first elements (Figure 64) that were designed for the Reality Editor are a variety of buttons. A button sends an on signal when tapped and off signal when released. The switch toggles between on and off for each touch and the flip switch keeps a single node active at a time. All buttons send out digital signals—they are at either 0 or 1.

For more granularity, a variety of sliders were designed (Figure 65). There is a normal slider that allows a user to slide multiple values between 0 and 1. A half-circle slider allows a better
understanding when controlling rotations of a motor, for example. If absolute control for rotations is needed, a full circle can be beneficial. The circle starts with 0° being equal to 0 and 360° being equal to 1. A 180° position equals 0.5 and so on. The circle can also extend into a two-dimensional input element, where the rotation provides one value and the distance to the circle center a second. Eventually a 2-D slider field allows a value input on the x and y axes.

![1 DoF Kinetic Slider](image1.png) ![2 DoF Kinetic Slider](image2.png)

*Figure 66 Illustration for AR kinetic sliders.*

Certain user interfaces require the user interface to extend beyond the visual boundaries of the interface element. For example, a feedback loop can be created where the motion of an object and the augmented reality user interface are synchronized. This could be, for example, a car that follows the position of the user’s finger on a screen. This means that if the finger touches the user interface element for a forward motion, the car would want to move until the AR user interface element center point reaches the position of the finger. For such interactions, a special kinetic slider model is required (Figure 66). Such a slider needs to be connected with the user’s finger for as long as the finger is on the screen. The Reality Editor provides such a kinetic slider for one degree of freedom as well for two degrees of freedom.

Beside elements that only provide outputs, there is also a set of interface elements that provide inputs.
Complementing the buttons and sliders, the Reality Editor implements single digital value visualizers (Figure 67). These can be seen as equivalent to LED lights. When connected they show the state of a single node. An analog value bar indicates the state of a floating-point signal. The same signal can be presented with a circular indicator and eventually a graph can print multiple values over time.

*Figure 67 Illustration for AR data visualizers.*
A combination of input and output allows visual feedback when a slider is operating a mechanical system with inertia (Figure 68). For example, consider a motor that only operates at a certain speed. If a user operates a rotation slider to a new position, such a motor might require some extra time to reach that position. For such cases, an output value is connected to the motor, and the position of the motor is sent back to the interface element via an input. The difference
between the user-set position and the position of the physical system is represented with a yellow placeholder. This allows the user a better understanding of actions taking place.

*Figure 69 Illustration for 6-DoF kinetic sliders.*
Additionally, there are interactive systems that require more than two degrees of freedom (DoF). For such specific systems, such as robotic control, another set of user interfaces is required (Figure 69). For these interfaces, not only the position of a finger relative to a user element is used, but a system of finger position on screen, as well the 6-DoF position of the mobile device that runs the Reality Editor, relative to the marker positioned on the robotic system. For experiments, a 6-axis robot was built with Lego Mindstorms is used to prototype the 6-DoF input control.

The user interface for 6-DoF works such that when the finger touches the user interface, the momentary translation and rotation position of the phone relative to the marker become the origin position. All changes of the relation between marker and phone become control vectors for the robotic system. The robotic system tries to reach back to the origin position and therefore when the origin position is reached, the control vectors are zero again.

The 6-DoF visual representation follows a simple model, where two circles overlay each other with an imaginary distance between them. The white circle is placed on top of the marker and therefore always stays at the same position relative to the marker. The green circle, however, appears to be hovering over the marker. This means that if the marker (and therefore the object) is rotated to the left side of the x axis, the green circle would move to the left side as well. The same accounts for the y axis. Once the robotic system has reached the origin position again, the circles are back at the same position. Rotation in the z axis is indicated with a round placeholder indicating the angle between the starting and present positions. Translations in the x and y axes follow the same concept as the kinetic 2-DoF interface. However, a dotted line always indicates the distance between the circle and the user’s finger position. Translations on the z axis use the same model as the rotations, with two circles that have a distance between them, where the further circle is placed at the marker position and therefore does not change the size relative to the marker. When the phone moves closer to the marker, the green circle becomes smaller, because the marker becomes bigger and therefore the white circle increases size equally. As it moves further away, the white circle shrinks and the green circle appears bigger.
5.8. Color Schemes

Throughout the development of the Reality Editor and its augmented reality experience, a multitude of visual problems emerged. Quite often, user elements were not readable when they were superimposed onto the video stream. Some experiments with colors, brightness and opacity of user elements showed that there are certain sets of colors that work better than others in an AR user interface. Every color that resembles natural colors competes with the colors found in the AR video stream. Therefore, AR user interfaces need to use colors that do not naturally appear in a video stream. Neon colors fit this requirement, as they are positioned at the edge of the color spectrum. For example, a blue color that is suitable for AR has full brightness in green and blue. The resulting cyan color is the least likely color in a normal AR user interaction. Therefore, all user elements in the Reality Editor have this color. Other colors are a fully saturated green, a fully saturated yellow and a specially chosen red, that departs from the full saturated red into...
purple in order to separate it from more common red tones. A white with maximum brightness works nicely as well. All user interface elements in the Reality Editor are made with these four colors and white, as shown in Figure 70.

Additionally, each color also has a specific use:

1. Blue indicates that the user element is interactive.
2. White indicates that an element is not interactive but contains visual information.
3. Green is used to present active data, visual feedback or general positive response.
4. Yellow is used for indicating visual feedback that represents actions between the user interface and the user or just temporary active data.
5. Red indicates a negative response or an inactive element.
6. Use Case Prototypes

While developing and testing the Reality Editor, I built and tested a multitude of prototypes. This chapter discusses each of these prototypes and how they contributed to the development of the Reality Editor. Initial experiments were followed up by some larger collaborations with members of the Media Lab consortium.

6.1. Lego as Physical Building Block

![Diagram of Lego block setup]

If sun is dark and door gets opened, switch on the light for 3 minutes.

A core idea for this thesis is how to enable a user to create little programs that help to control how physical objects are connected among each other. For a full set of expressions such a virtual program, virtual logic block or logic nodes should have physical counterparts. One could build such blocks from scratch, but there is already a physical programming block system that has been proven to work—Lego Mindstorms provides a variety of sensor and actuator blocks. The Reality Editor shares a lot of philosophical overlaps with Lego, as one Lego provides a simple block that combined with others can build possibly unlimited varieties of objects. Yet each block provides enough guidance for a non-expert to build interesting objects. The right balance
between guidance and free form building is a key to the success of Lego toys. The Reality Editor seeks to build a similar balance between guidance and free-form programming. The entire Lego set is modular in nature. The modularity finds a perfect overlap with the Lego Mindstorms toy. Ideas that have physicality and virtual logic can easily be modeled with virtual Reality Editor building blocks and physical Lego Mindstorms blocks (Figure 71).

![A screenshot from the Reality Editor showing an early node-based programming interface on a robot.](image)

A first Lego Car (Figure 72) was built to test the feasibility of using Lego Mindstorms as physical building blocks for an augmented reality programming system that would seamlessly combine physical and virtual building blocks. The Lego Car was also used to test some first web experiments. The idea was to build the Reality Editor server into the limited processing capability of an embedded microprocessor. For this purpose, a 32-bit ARM MCU was combined with the TI-cc3000 Wi-Fi chip. An entire HTTP stack was written for real-time data communication and connected to a Reality Editor server running on a computer. A custom circuits was used to connect the Lego motors and sensor with the MCU. This first experiment implemented the virtual logic nodes as UI elements that a user could drag and drop onto the connected object. Once the virtual logic nodes were released onto the connected object, a user could set the state of the logic node via a ring surrounding the node.
What worked very well with this approach was the simplicity of wiring as a small program onto a physical object. In particular, the simple drag and drop method that dropped virtual nodes onto the physical connected object appeared intuitive. However, the downside of this interface was that it became difficult to extend the program to anything more complex. It took up too much of the visual screen space. On top of these problems, the settings rings around the node interfered with the virtual patch wires, which resulted in many false inputs. On the technical side, the HTTP stack was so low level that it became difficult to implement many ideas in this demo.

The Lego Car also implemented individual user interface components (Figure 73). Every sensor had a small progress bar interface showing its value, and every motor had a virtual joystick that allowed forward and backward movements. Every sensor or motor was treated as an individual connected object. Thus, the interface exposed to a user was the result of how the user combined the physical Lego bricks. The Lego Car could entirely change its shape and the virtual input and output elements would adapt.

The experiments with Lego showed that there was potential, but other solutions for programming the nodes as a user, as well as for programming the server infrastructure with a higher-level programming environment, were required.
6.2. Smart Grid

The smart grid interface experiment (Figure 74) focused on a visual representation of a smart grid application meant to help users optimize their power consumption throughout the day. A virtual ring with 24 tiles circled around a power outlet. The ring represented the 24 hours of each day, with the actual hour always staying on top. Each tile showed a color representing the estimated electricity cost for that hour of the day. A red tile represented a high price, while a green tile represented a low price. The user would tap on the tile showing the highest acceptable price. All of the tiles with the same or a lower price would be marked within a green segment of a second ring, while the segment around the rest of the tiles would be marked red. The colors of this second ring represented the switch state of a Reality Editor programming node.

If a user’s appliances include the Reality Editor server, they can be wired up to the smart grid interface. For example, if a user wants to start a washing machine, a virtual patch wire is drawn from the smart grid interface to the start button of the washing machine. Once the electricity reaches the right price, the smart grid interface switches its node on and sends the washing machine a signal to start.
6.3. Health

Figure 75 A screenshot from an augmented reality experiment that shows a temporary tattoo used as a marker.

The health demo (Figure 75) was created to explore how the human body could connect to the physical environment. With this demo, a user could focus the Reality Editor on a temporary tattoo on the forearm and see real-time health data such as their heart rate, temperature, and galvanic skin response (GSR). The tattoo was placed on the left arm and the sensor readings came from a sensor placed on the right arm. Every sensor reading had a node to connect to the Reality Editor ecosystem. For example, a user could connect their heartbeat readings to the brightness of a light in the room so that the level of illumination would reflect the user’s pulse. This generated an interesting amplification of something that normally is very personal. Another experiment connected the user’s GSR to the color of the light, allowing the color of the room to reflect the user’s stress level. It became evident that experiments could be conducted with the Reality Editor very quickly and easily. Whereas before researchers would need to write new software for every experiment conducted with a health sensor, now they would only need to
provide general interfaces that could be connected. Hardware and software components could be created once and easily reused for other experiments.

6.4. Museum prototype

The Reality Editor is a web browser at its core. This means that web information and services can seamlessly be integrated into the physical world. The museum demonstration (Figure 76) was created to experiment with these capabilities of the Reality Editor. Different social sharing buttons, a Twitter feed, and an online music player were used to support the visitor experience in

Figure 76 A screenshot from the Reality Editor showing artwork augmented with web services.
a museum. The music player and the Twitter feed work well. However, since the Reality Editor itself is a webpage that loads content into iFrames, it has some difficulty dealing with web content that loads new page content with a different format. For example, the social sharing buttons link to a second popup page. As popups and new frames are not supported, the share buttons are not fully functional. Importantly, the Reality Editor is capable of loading all of this content at the same time.

6.5. Lights demonstration

Figure 77 A screenshot of the Reality Editor showing the interaction with a general-purpose knob and a light.

An overall theme used in prototyping the connected objects are lights and general-purpose buttons and rotation knobs, as shown in the light demo (Figure 77). Since these interfaces never changed and the protocol never changed, they showed how stable the Reality Editor system became as it developed. The user interface and the light emerged as a collaboration with the
Media Lab member Steelcase as a general idea to control modular control the office environment.

Although the light demonstration appears simple, it turned out to be the perfect testbed to prototype new interface behaviors and use cases. For example, the first Reality Editor iteration used color coding to indicate the origin and destination of a virtual patch wire. However, when implementing the logic nodes, the color of the patch wires suddenly became important for another functionality. Since the color coding was unintuitive in the first place, another solution for indicating the data flow was needed. The final iteration showed the patch wire as a row of dots that flow in the same direction as the actual data is flowing.

6.6. Arduino demonstration

![Arduino boards](image)

*Figure 78 A screenshot of the Reality Editor showing two Arduino Yun boards being connected.*

One of the earliest ideas for the Reality Editor was to enable the creation of connected objects with the Arduino programming language [85] alone (Figure 78). Since Arduino is used by many product and interaction designers, it would allow them to work together to build new connected
object prototypes. This is important, because often the engineering approach is different from the design approach. Further a designer can dive into the prototyping stage without the help of an engineer, more of a designed experience with connected objects is possible.

The Arduino Yun was targeted as a development environment because it had a capable system-on-a-chip as well as the normal Atmel MCU required by the Arduino compiler. However, the software preparation for the Arduino Yun was very difficult. The user needs to download a special image file that has to be cloned on to an SD card, then log in to the Arduino Yun and configure network settings and device names. Only a limited number of people have been able to use it. What started as an idea for designers eventually became more of a project for geeks. However, this prototyping setting was still the easiest way to allow others to build connected objects using the Reality Editor. The Arduino Reality Editor combination provided others with a possible starting point and even led to the creation a Master’s thesis by Carsten, with the title “Evaluation of an Augmented Reality Operating Concept for Smart Home Applications” [86]. Carsten used the Reality Editor and the Arduino Yun board to perform studies on visual programming and introduced technical further developments as part of the open source community. For example, Carsten helped to implement the first hardware interface modules, and also implemented an initial method for display overlays that would provide visual feedback.

### 6.7. Audi Car demonstration

For a real-world test outside of the laboratory, two scenarios were created: a common work desk scenario (Figure 79) and a car scenario (Figure 80). Since both scenarios were built with the Reality Editor, all use cases are seamlessly compatible. For the office scenario, the previously described desk lights and general-purpose knob were used. In addition, a chair was created that can measure whether a user is sitting in it. This measurement is represented as a node. A second node was created that represents the same measurement as a delay switch.
In a second scenario, the functionality of an Audi A4 automobile was modified to be compatible with the Arduino Yun. Through these modifications, functionality such as the air conditioner, the window positions, and the audio volume were controllable via the Reality Editor platform. Two general-purpose knobs were added to the dashboard next to the radio, to be used with the Reality Editor platform.

The goal for this scenario was to learn how to provide a user with a simple way to reprogram the functionality of physical interfaces in the car, allowing for the operation of flexible physical interfaces while driving. At this point, such a degree of flexibility can only be provided with the
use of touch screens. However, a touchscreen is less than safe to operate while driving. Our platform provided significant value by enabling flexibility in a tangible, safe interface.

![Diagram](image)

*Figure 81 Four nodes that represent windows in a car are daisy-chained so that they execute the same signal input.*

It was interesting to learn how quickly both scenarios could be created in software, once the hardware technology was working. After they had been implemented, it was possible to just playfully explore different possibilities. For example, the office chair could be connected to the lights to provide a personalized automated light switch (the light switched on when a user was detected in the chair). The desk chair could be connected to the car air conditioner to automatically lower the interior temperature of the car when the user left the office. The general-purpose knob could control the lights and some functions within the car. A daisy chain of multiple nodes that represent the car windows (Figure 81) allowed the user to operate all 4 of the windows with a single button. Eventually, the freeze functionality of the Reality Editor was used to make a still image of the car’s AR GUI so it could be used as a remote control to control the car from a distance (Figure 82).
6.8. Virtual Objects

Another experiment allowed the inclusion of 3D content into the Reality Editor. Since the Reality Editor is a web browser, WebGL can be used to include content. Two new APIs were added for this demonstration. The first allows every user interface to become full screen, while the second allows an interface to subscribe to its own projection and transformation matrices. This allows the interface to take over the 3D transformation. As a result, the WebGL rendering treeJS can be used to render 3D objects as augmented reality content into the Reality Editor (Figure 84). Since these 3D objects are rendered in the Reality Editor, all functionality possible with the Reality Editor is enabled with the 3D objects as well. For example, a physical general-purpose knob can be used to change the colors or size properties of virtual 3D objects (Figure 83). This demonstration could become interesting when prototyping animations for movies or testing physical machines virtually before the physical objects start moving.
Figure 83 With the crafting mode, a user connects a physical rotation knob to two nodes that are placed on a marker. Each of these nodes represents one of the digital properties of a teapot.

Figure 84 Once the connection from Figure 83 is created, the user switches to the object interaction mode where a 3D teapot is visible. Operating the physical rotation knob changes the shape of the teapot.
6.9. Lego Mindstorms prototype

With all of these experiments, two foundational components of the Reality Editor became stronger. First, the node-based programming stayed consistent and gained more and more capability. Second, more and more APIs were created to provide the web-based user interfaces with more functionality.

The original Lego Mindstorms demonstration that started the idea of physical and virtual programming blocks was too limited for intensive tests, so it was time to build something that would really test the capabilities of the Reality Editor. The Audi demo showed that the Reality Editor could be deployed in a real-world object; the goal of this new Lego Mindstorms demo was to show how complex such a deployment could become. A secondary goal was to show the capabilities of the newly integrated hardware module API. The robot that was built (Figure 85) has six movable axes and four-color sensors to help with calibration. The entire system was constructed using separate units that follow the Reality Editor node paradigm. Eventually the control of the entire robot was programmed with the Reality Editor virtual patch wires.
The robot is operated by holding the Reality Editor onto one of the joints. Once the interface appears, the user can lock in translation and rotation of the phone relative to the robot joint. This means that whenever the user moves the phone with the Reality Editor on it, the translation or rotation vector between phone and robot joint changes. The vector is sent to the robot for motor control. The motors are programmed in such a way that the robot tries to move the joint back to a position that resamples the original locked relative position between phone and robot joint.

This demonstration proved the following:
1. The Reality Editor server enables a distributed real-time feedback loop between AR interface, physical connected object, and networking infrastructure.
2. The Reality Editor programming interface is capable of programming complex machines
3. The web-based interface is capable of real-time machine interactions.

6.10. Lego WeDo prototype

From the start the Reality Editor was built to be shared with an open source community. In the beginning, this was thought to be designers and engineers. When published, however, the Reality Editor gained a lot of public attention, and not everyone who was interested could really use it. Specific know-how such as building interactive webpages with JavaScript and HTML and how to clone disk images and configure a Linux system was required. The goal of the second implementation, which included new APIs, the logic crafting interface, and the instant user interfaces, was to lower the barriers of entry significantly.

Two new hardware interfaces were created to include the Lego WeDo robotic education set [REF] and Philips Hue lights [REF]. Both the Philips Hue and the Lego WeDo contain buttons that allow an instant connection. A stand-alone OS X app (Figure 86) was created that would provide a graphical user interface for these two hardware interface modules. As a result, a user can download the OS X app, open it, and then push the connecting button on the Philips Hue hub or the Lego WeDo block. The objects connect instantly and can be controlled with the Reality Editor smartphone app available in the iOS app store.
The Lego WeDo demonstration (Figure 87) allows everyone to instantly experiment with augmented reality programming. As such, it allows a user to create a program with physical and virtual building blocks and associate it into a room or a toy.

Figure 86 OS X standalone application for serving the Reality Editor mobile app.
Figure 87 A screenshot taken from the Reality Editor interface that shows two Lego WeDo 2.0 blocks with superimposed directly mapped nodes for all attached and built in components. Beside a built-in button and LED light, each Lego WeDo 2.0 block has a distance sensor and a motor attached. The shown program connects a button to a logic node and a LED light. The logic node is connected to a second LED light. The logic node is programmed to invert the signal, so that when the button is pushed one LED light starts glowing while the other stays dark and vice versa for when the button is not pushed.

6.11. Secure Transactions

This entire thesis has focused on the possibilities that come with using augmented reality for programming physical objects. However, some interesting use cases come up when the Reality Editor is seen as an augmented reality web browser. Since a web browser is made for secure online transactions such as banking, the Reality Editor inherits these same capabilities.
In one user interface demonstration (Figure 88), the Reality Editor is used to perform a just-in-time payment transaction for a parking meter. This demonstration is more for inspiration than for actual use, because it was not possible to implement the functionality for paying for a parking spot in Cambridge. However, the demonstrated system was able to establish a HTTPS communication to a remote server and communicate the selected payment. Therefore, this scenario is feasible. In a real-world scenario, a user would register with a payment service. That payment service would keep the user logged in (as in a normal web browser) and perform the payment transaction when the user clicks the pay button on the augmented web interface. The right web interface can be detected using geolocation.

![Screenshot from a secure payment transaction.](image)

**Figure 88 Screenshot from a secure payment transaction.**

### 6.12. Visual Search

As a web browser, the Reality Editor comes with a rich data-consumption and rendering capability. The visual search demonstration shows how powerful the Reality Editor is compared to other augmented reality solutions such as Unity 3D. A line of experiments with Media Lab
member Target were tested and implemented on backend systems from Target. This means that a working prototype could instantly be transferred into a product development stage.

The visual search is particularly useful. A user enters food preferences into the Reality Editor (Figure 89), then points a device at products in the store. The Reality Editor shows instantly whether the products match the user’s preferences. A user can walk through an entire store and potentially browse thousands of products in very little time (Figure 90). The data for this browsing experience are loaded from a remote back-end server, since the entire system is web-based. As such, a retail store can contribute to the user’s browsing experience by curating web content, the same way web content is provided for millions of clients via a normal web browser.

Figure 89 A user selects personal preferences for the spatial search.
Figure 90 The Reality Editor displays spatial search results.
7. Workshops and Community

Workshops and interactions with the open source software community were integral to validating the research presented in this thesis, as described in this chapter.

An initial workshop to test the Reality Editor interface and its use with the Arduino platform was performed on February 14, 2015. The first version of the Reality Editor was released on GitHub on June 21, 2015 [87], complemented with a talk at the Solid conference in San Francisco [88], and a website that included a forum [89]. The first workshop for the open source community working with the Reality Editor was held September 2 and 3, 2015. In December 2015, a video about the Reality Editor was released to the public, to further encourage the open source community to contribute to the development of the Reality Editor. A second workshop was held March 3 and 4, 2016.

The second version of the Reality Editor was published on GitHub on May 21, 2017 [90], complemented with a talk at the AWE 2017 [91], a redesigned website and forum [92], a stand-alone application that allows users to test the Reality Editor with a Philips Hue light [2] or the Lego WeDo 2.0, and a new video.

7.1. First Workshop

The first workshop included seven participants, and focused on understanding whether the Reality Editor user interface and the creation of connected objects with the Arduino platform met expectations for simplicity. The participants were introduced to the Reality Editor platform and each was provided with a prepared Arduino Yun and an iOS device with the Reality Editor preinstalled. Half of the participants were engineers, and the other half were designers. Only one of the participants had seen or used the Reality Editor before.

After the workshop, the participants were asked to answer the following questions:
How difficult did you find the creation of an object? What are your suggestions for future versions?

Some participants said that it was not particularly difficult to build a connected object, but that there were too many steps to follow. Almost all of the participants said that they would have trouble building connected objects without a tutorial. One participant said that the most complicated part of the process was creating a new augmented reality marker. This step used the Vuforia AR tracking solution [52], which requires a user to log in to an online system and carefully follow a number of different instructions to generate a marker.

When asked for suggestions for future versions, all of the participants wanted a simpler user interface and a better way to create targets. The best possible method for generating a target would be dragging and dropping an image onto the server, but there is currently no AR tracking technology that provides an API allowing for such implementation. Two of the participants also requested drag-and-drop, just-in-time user interfaces that would allow a user to add a new user interface to a connected object without programming HTML. This request was implemented in a later Reality Editor version and described in chapter 5.7.

7.2. First Release and the Web Forum

Publishing the source code and making it freely available was an important design consideration. Building an open source community around the Reality Editor was therefore part of this thesis from the beginning. While literature was helpful in determining how to engage an open source community around the project [93], but the most valuable information was drawn from studying examples such as the Open Frameworks community [47] and the Arduino community [85]. Open Frameworks uses a very good forum service called discourse [94], which also became important to the Reality Editor community, and GitHub itself provides amazing tools for shared software development. The first webpage was built with Adobe Muse.
The forum developed as expected, starting with a few developers who would post general questions and offers to help, and expanding after the first video was published and attracted media attention. In general, it had three kinds of users. There were developers who wanted to support the project, developers who wanted to create augmented reality experiences, and eventually users who knew just enough to build a webpage and who wanted to learn more about the Reality Editor and augmented reality. All of these groups helped solve problems at different scales and find bugs.

As this thesis is being written, the forum has 254 registered users who have written 1,200 posts and accumulated 27,500 page views. Unregistered, anonymous users account for another 199,100 page views.

These are the most viewed topics:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Views (k = 1000)</th>
<th>Replies</th>
<th>Titles (created by users of the web forum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.2k</td>
<td>71</td>
<td>“First step to install Open Hybrid on raspberry pi?”</td>
</tr>
<tr>
<td>2</td>
<td>4.6k</td>
<td>83</td>
<td>“New hardware interfaces API”</td>
</tr>
<tr>
<td>3</td>
<td>4.6k</td>
<td>22</td>
<td>“Displaying data collected from an external Database”</td>
</tr>
<tr>
<td>4</td>
<td>4.0k</td>
<td>93</td>
<td>“New Reality Editor Version 1.5.7”</td>
</tr>
<tr>
<td>5</td>
<td>3.7k</td>
<td>29</td>
<td>“ofxQCAR for android”</td>
</tr>
<tr>
<td>6</td>
<td>3.5k</td>
<td>57</td>
<td>“Openhybrid Raspberry pi - interface and library”</td>
</tr>
<tr>
<td>7</td>
<td>2.7k</td>
<td>11</td>
<td>“ARToolKit 5 add-on for Open Frameworks (iOS &amp; Android)”</td>
</tr>
<tr>
<td>8</td>
<td>2.5k</td>
<td>25</td>
<td>“Documentation - Work in Progress”</td>
</tr>
<tr>
<td>9</td>
<td>2.2k</td>
<td>17</td>
<td>“Reality Editor proof of concept video”</td>
</tr>
<tr>
<td>10</td>
<td>2.1k</td>
<td>2</td>
<td>“How to connect Arduino with Lego Mindstorms”</td>
</tr>
<tr>
<td>11</td>
<td>2.0k</td>
<td>24</td>
<td>“Targets other than image targets”</td>
</tr>
<tr>
<td>12</td>
<td>1.9k</td>
<td>10</td>
<td>“Creating Unique AR Interface Elements”</td>
</tr>
<tr>
<td>13</td>
<td>1.8k</td>
<td>19</td>
<td>“Using Open Hybrid as front end”</td>
</tr>
<tr>
<td>14</td>
<td>1.8k</td>
<td>15</td>
<td>“Real Time monitoring error”</td>
</tr>
<tr>
<td>15</td>
<td>1.7k</td>
<td>18</td>
<td>“H/W Config Editor”</td>
</tr>
</tbody>
</table>
The forum was a valuable tool for understanding the limitations and advantages of the original Reality Editor implementation. All of the most-viewed topics listed above were created before the release of version 2.0 of the Reality Editor, and many led directly to improvements. For example, the community’s interest in using Raspberry PI with the Reality Editor was unexpected. After a discussion about the requirements, someone from the community started to research how to port the software code from the Arduino platform to the Raspberry PI and eventually provided instructions and a hardware module as well. Thus, Raspberry PI support for the Reality Editor grew organically from the community’s interests.

The Raspberry PI required a different hardware interface than the Arduino platform. When the software was first released, the intention was that the hardware interface would allow a multitude of different systems to be compatible with the Reality Editor. However, the development of such a hardware interface was far from complete. This was the focus of the second most viewed topic. Together with the community, an outline for the hardware interface modules was formed and then implemented with a community approach. A particularly important finding was the necessity of massive modularization. Many community members contributed new hardware modules, because the API for these hardware modules was simple and well defined. Modularity meant that everyone could join without the need for complicated coordination with the rest of the community. The lessons from the hardware interface module development were adapted to reorganize the source code, and also led to the idea that the logic crafting programming blocks should be highly modularized as well.

Besides the development of new software, many questions related to building user interfaces. For example, the third most viewed topic was about displaying external web content. The Reality Editor specified how to communicate data between user interfaces and the server infrastructure. However, many users started to use it for other applications, because the HTML prototyping allowed them to easily experiment with augmented reality. The solution for this problem was easy. The Reality Editor server delivers an ordinary webpage to the Reality Editor, and an iFrame placed into it any external web service to be embedded.
A number of topics were related to the availability of the Reality Editor on the Android platform. A problem in the C++ libraries for the Open Frameworks programming environment with the Vuforia AR environment [52] prevented compiling for the Reality Editor on the Android platform. The community took a range of approaches (topic 5, above) to solving the problem, but no solution has yet been found. From a research perspective, this problem was not particularly important. Supporting only the iOS platform allowed the Reality Editor to be released with a minimum of difficulty. The iOS platform is very consistent, and runs on only a few devices. Still, the Reality Editor had to be tested on almost all of the iOS product lines, because each one had slightly different camera tracking and screen settings that would complicate the openGL to CSS3D translation or corrupt the video stream. The Android platform, with all of its different vendors and phone hardware, is a nearly impossible environment for maintaining a small AR project.

7.3. Second workshop

The second workshop, which lasted two days, was planned to give the growing open source community a platform for experimentation and learning. The first day was focused on introductions to the overall platform and examples from the participants, while the second was dedicated to building ideas with the Arduino Yun and the Reality Editor platform. As the workshop was held before the platform got a lot of public attention, the participants mainly came from the MIT and Media Lab communities.

One of the questions this workshop was intended to answer was whether the processes for programming the Arduino Yun boards were easy to understand. The result was nearly disastrous. Everyone came expecting to build, but it took two to three hours to teach everyone how to prepare the Arduino Yun boards with the right firmware. Once the firmware was installed, however, the participants had similar experiences to those of the first workshop. This workshop made it clear that the Arduino Yun platform is too complicated, and that not only the Arduino Yun boards but also the AR marker files needed to be pre-produced, so that participants do not have to carry out all of the complicated steps required by the Vuforia marker generation process.
Another interesting exploration in this workshop was that participants started to use the ornamental codes introduced in chapter 4.6. A code generator for these ornamental codes is publicly available as well. Unlike the previous day, members of the community adopted these codes quickly, and incorporated them into their projects, as did designers of other Reality Editor workshops. Either these codes generate a general identification of the Reality Editor or they are easier to handle than other marker images. Identification means that a user recognizes an ornamental code as a visual code that provides augmented reality content for the Reality Editor. Both cases support the intention behind generating these ornamental codes.

7.4. Third Workshop

The third workshop was a special kind of event. The applications for the third workshop opened after the Reality Editor had been online for some time and the new community had gained extra publicity through multiple news outlets writing about the Reality Editor. Around 20 participants applied for the workshop, but there was a problem created by the intersection of a global community and a free workshop. Many of the applicants were members of the international open source community, and only a few were able to travel to the United States and participate in the workshop. Eventually, five participants came to share their work and learn about the Reality Editor platform, but some of these were more interested in visiting MIT than in participating in a workshop. As a result, this third workshop did not provide a good basis for drawing conclusions about the Reality Editor. However, the application included a survey intended to discover more about the interests of the participants. As many of the applications came from users who had used the Reality Editor for their own experiments, this online survey gives interesting insights into the Reality Editor development community.

The applications came from a mix of designers, engineers and scientists. Their interests were wide-ranging, from “functionality that could serve the kids, the seniors and the disabled” to a focus on IoT infrastructure. When asked about the positive aspects of the Reality Editor platform, almost all of the applicants emphasized that it made building augmented reality...
applications easy. One participant said that not only was its simplicity terrific, but so was its ability to build technology on top of the Reality Editor platform. The cumbersome Arduino Yun was the main negative. Several users also wanted less coding and more interaction.

The workshops and the applicant survey led to the following conclusions:

1. The Arduino Yun is a very good platform for programming interactions. However, it is difficult to set up and too complicated for designers.
2. Generating AR tracking markers with the Vuforia framework is too complicated.
3. The Reality Editor provides a simple tool for generating augmented reality experiences with HTML.
4. Modularizing the contribution framework encourages the open source community to contribute code to the Reality Editor project.
5. Users accept the ornamental codes, and they are effective as visual identification markers for augmented reality.

7.5. Royal College of Art Workshop

After analyzing the previous research findings from the open source community interactions and workshops, a final workshop was planned to explore whether under the right conditions product designers could build augmented reality experiences and connected objects without the need for engineering help. This workshop was facilitated together with the Royal College of Art Design Products program in London.

The workshop was scheduled over a period of four days. In addition to the workshop days, the product design students were given a homework assignment to bring one existing object that could become connected, and a concept drawing for an object of the future.

The first day started with introducing the objects and concepts, followed by an introduction to the Reality Editor platform. The students then formed groups for generating ideas until the end of
the day, when each student had the chance to present multiple ideas in a two-minute pitch. The group voted to find the best ideas for well-distributed teams.

The second day started with a hands-on session where the students learned to build a simple example with the Reality Editor platform. All of the Arduino Yun boards were preprogrammed and all of the markers were pre-generated so that the students could focus on creation. Once the students gained the knowledge to build content for the Reality Editor, they entered a project conception phase that lasted until the end of the day. The following two days allowed the students to build working prototypes of their ideas, with a final presentation in front of the product design department.
The following demonstrations were built by product design students who participated in the workshop. None of the students had experience with augmented reality before the workshop.

7.5.1. Bookmark by Fiona O'Leary

![Figure 92 Connected bookmark by Fiona O'Leary. The bookmark can sense whether a book is closed.](image)

The connected bookmark (Figure 92) is a simple yet powerful object that uses a sensor to detect whether it is in a closed or open book. The sensor is connected with an Arduino Yun that connects the bookmark with the Reality Editor platform, allowing it to control an infinite number of connected objects. For example, a user could connect the bookmark to a door lock and a lighting system, making it possible to lock the door and turn off the lights when reading in bed, by simply closing the book before falling asleep.
7.5.2. Pill Box by Václav Mlynář

Figure 93 Augmented Reality Pill Box by Václav Mlynář. The pill box can sense when new pills need to be ordered and provide the user with options for purchasing them.

In Europe, pills are usually sold in branded packages or boxes, unlike the pill bottles filled at the pharmacy that are more common in the United States. Because such pill boxes distinctly identify the medication inside, an augmented reality interface can be used to provide additional information. Each time a user takes a pill, the pill box can detect how many pills are left in the box and provide the user with messages when a new package need to be ordered. By simply clicking a virtual button click on the pill box interface, a user can order a new box.
7.5.3. Card Key by Kyungmin Han

Kyungmin’s card key tries to solve a particular problem. In corporate or university buildings, a user may have access only to a subset of rooms, which could be scattered throughout the building. This can make it difficult to understand which rooms the user can access. A user can point the Reality Editor at this key card to see a plan of each floor of the building. Green zones indicate the rooms that can be accessed. As the user moves the keycard closer to or further away from the phone, the view shifts to other floors. For example, if the ground floor is visible, moving the key card closer to the phone will reveal the second floor. While Kyungmin developed the interface elements, an API needed to be added to the Reality Editor to read the distance between the phone and the marker. This simple add-on later grew into a full, complex API.
7.5.4. Augmented Communication by Kristian Knoblauch

![Image](image_url)

*Figure 95 The user points a Reality Editor at objects that have an ornamental code sticker attached. Object-related information is displayed.*

The augmented communication demo by Kristian Knoblauch makes use of the HTML capability of the Reality Editor. Online services and videos are embedded in speech bubbles linked to physical objects decorated with ornamental codes that the Reality Editor can use as tracking markers. The ornamental codes also signal to the user that the object contains augmented reality information. Knoblauch’s demo validates the original intention for creating the ornamental codes, and also shows how easy it is for a designer to implement web technology in augmented reality applications.

The RCA workshops proved that product designers who have no previous experience with augmented reality technology can realize their ideas using the Reality Editor platform. It also proves that once the user is relieved of the Vuforia marker generation and the cumbersome setup of the Arduino Yun, the workflow of the developer tools described in this thesis meets expectations.
The current implementation of the Reality Editor removes specific support for the Arduino Yun and replaces it with a general Arduino hardware interface that works with all Arduino models.
8. Discussion

Up to this point, this thesis has introduced a number of new ideas and expanded the boundaries of augmented reality and connected objects. However, there are still many questions. This chapter discusses a few of the most important ones.

8.1. Reality Editor: A Critical Discussion from the Perspective of Affordance

How do we interact with objects and how do we learn to do it? James Gibson [7] characterizes the relationship between humans and objects as affordance. Affordance describes the symbiosis of performable actions involving the user, as well as the physical objects with which the actions are performed. For example, a human user can pick up a small stone, because it is the right size to fit in a human hand. A larger stone might be the same shape, but be too heavy to pick up or too big to hold comfortably. Therefore, it is important to look at the object and the user at the same time.

Gibson writes: “The affordances of the environment are what it offers the animal, what it provides or furnishes, either for good or ill. The verb to afford is found in the dictionary, but the noun affordance is not. I have made it up. I mean by it something that refers to both the environment and the animal in a way that no existing term does. It implies the complementarity of the animal and the environment” [96].

Don Norman, on the other hand, explores the concept of affordance in a less generalized way. In his book *The Design of Everyday Things* [44], he explores affordance as the way things are used and how errors occur. He claims that there is no human error—only incorrect design. For Norman, affordance is not only a result of the possible interactions between humans and objects, but also the human perception and chains of possible interactions. For example, a door could have a handle that users can either push or pull, without indicating which action the user should take. However, the door handle can be designed to indicate in which direction the door opens,
thus eliminating a common door-operation error. Norman’s concept goes beyond that of Gibson, because it involves cultural knowledge as well as perception into the actual use of objects, not just the physical possibilities for their use.

In my research, I have explored a new concept of connected objects [97], which are physical and virtual at the same time. These objects provoke a number of critical questions. How do we learn to use them? What affordances do they have? How do we become aware of these affordances? These questions also highlight another interesting aspect of this work. The shapes of physical objects tell us how they can be used. These shapes can be perceived simultaneously by all of the members of group, and they can have social affordances [98]. In the context of interaction, such social objects have the property of self-evidence, fully communicating how they can be used. The concept of self-evident things is well-explored by Durrell Bishop [99].

Most electronic devices are not social objects. They are complicated, opaque boxes that do not readily reveal how they are operated. To use them, one needs to read a manual or text displayed on the screen. The actual operation then relies partially on an abstract chain of consequences in one’s own mind. This makes them very personal objects, as their use and meaning may vary from one user to the next.

Self-evident things are different. They explain their use through their visual and tactile properties, as discussed in the following section on connected objects from the perspective of affordance and self-evidence.

8.1.1. Affordances of Connected Objects

Figure 96 shows two common, physical, electronic objects—a toaster and a food processor. The toaster has the affordances of being “stick-bread-into-it-able,” “push-a-slider-down-able,” and “rotate-a-timer-knob-able.” The food processor has the affordances of being “pour-fluid-into-it-able” and “rotate-a-knob-able.” As a consequence of these affordances the toaster toasts bread and the food processor mixes dough.
Since these two objects are fully physical and all of their affordances are directly exposed, these objects are also self-evident. At any given time, a user is able to easily discover how these objects are to be used, and learn about the object by simply operating it and observing the consequences. This is called a constant direct feedback loop. Pushed buttons or rotated knobs have direct feedback that can be perceived with human senses. The objects are also designed in a way that explains their actions. A bowl that can be filled with a fluid is shaped so that the fluid flows to the bottom of the bowl. The slider on the toaster is arranged so that the toast slides down into the device as pressure is applied to lever. A clicking lock mechanism provides feedback signaling that the toast has found its rightful position. Once a user understands these interactions, he or she can memorize them and trust that the devices will always respond to the memorized patterns in the same way. These patterns can also be explained to other users, since the model is static, meaning that these objects will never change their functionality and a similar result can be expected every time they are used. Such static interactions can also be used as a reference in abstract verbal communication, where the objects are not physically present, and in collaborative situations where two users need to rely on known and defined objects so that the task they are performing will attain the expected result. The static nature of an self-evident object is what makes seamless interactivity possible for multiple users.

Figure 96 A toaster and a food processor.
Connected objects include both physical and virtual properties. The nature of the virtual properties means that physical properties that would normally be static can be changed. Therefore, a discussion of virtual affordances leads to a better understanding of both properties.

The best-known virtual affordance is the desktop metaphor [100], in which icons, windows, trashcans, and other virtual items represent analogies to the real-world work desktop. These analogies help us understand abstract data constructs. Furthermore, the icons become self-explaining or self-evident. For example, a trashcan represents the function of deleting files. The file itself is represented as a written page or, for an audiovisual file, an object that looks like a strip of film. The desktop metaphor (Figure 97) enabled non-experts to master the computer.

However, these icons and windows do not represent the kind of affordances Gibson had in mind. They do not rely on the possibilities of interaction between the user’s body and the physical space. Rather, they mediate an interaction with the user’s mind, while the user’s hands operate the same objects, regardless of the context—keyboard and mouse, or touchscreen. The desktop metaphor succeeds because of the physical world we live in; the properties we learned working
with real-world desktops allowed us to make an abstract transfer to the virtual world. But this transfer brings its own problems.

Every desktop is a personal intellectual construction. We organize, position, and personalize our virtual desktops, creating folder structures that only we understand. This means that the desktop is a single-user experience; multiple users cannot work on the same desktop without losing the flexibility of the desktop metaphor. Explaining the structure of the desktop to another person would require that it remain static. If one user changed the structure of the desktop, the next user would need to learn the new structure. Therefore, a parallel model of desktop multi-user interaction was developed. Users can log into the same computer, but each user has an individual desktop—a parallel desktop stored in the same physical location. As their virtual worlds never collide, the users never have to agree on common rules for working on the same machine.

Imagine building a model of the virtual desktop in the physical world. The result would be puzzling. Humans would be unable to interact with each other in the same physical space. Instead, the computer would always need to generate alternative realities with personalized affordances. Eventually, a collision of two or more realities would provoke a chaotic tangle of conflicts. What works for a completely virtual environment, therefore, cannot work in a connected object environment. As long as an object has a physical property, the entire chain of interaction needs to be a single process. Humans need to find paths to consensus to protect the integrity of the reality around them.

Thus, changing the properties of an object becomes a primary problem of human interaction, especially when multiple people are interacting with the same object. How does this influence the way we operate connected objects, which are physical and digital at the same time?
Let’s look back at the toaster and the food processor from a connected object perspective. In Figure 97 we can see that the toaster now has a virtual user interface, which could indicate whether a slice of bread is presently in the toaster. It could also be used to draw a pattern on the virtual representation of the toast, which could then be burned into the physical piece of toast. With the food processor, an augmented user interface could allow a user to draw a processing curve describing when the motor would need to work the hardest, allowing for more specialized recipes.

This function would not be easy to provide through either a physical user interface or a desktop computer interface. Combining a virtual user interface with the physical object, however, makes the same kind of sense as the slider that both moves the bread down into the toaster and indicates the current position of the bread. As long as a user can see the virtual augmentation of the connected object, these augmentations provide visual affordances for performing actions that are similar to the affordances Gibson writes about. By making the object even more self-evident, they provide a benefit for the user. Once a user actually begins to interact with the augmented user interface, Gibson’s strong physical relationship between the object and its use loses its meaning, but Norman’s concept still remains.

However, unlike the physical push slider, the virtual user interface is not directly available to the user. At the moment the augmentation disappears, the added functionality vanishes. Therefore, a
user would need to specifically know that opening an app on a mobile device and pointing it at the toaster makes this functionality available. Without this information, if a user makes changes to the settings, the next user might operate it without checking the augmented interfaces. With a toaster, this could lead to improperly toasted bread, but with a food processor, it could result in severe injuries. The question, therefore, is how an augmented interface for connected objects can be designed so that the operation of the physical user interfaces does not lead to misunderstandings when multiple users operate the same object.

**Option 1:** A user can change the setting of an object only for the next task unit.
For the toaster, this would mean that the default setting allows full operation with the physical interface, while the virtual interface is treated as a supplement that adds limited additional functionality. For example, a user would draw a specific pattern and the toaster would retain it for a specific amount of time or for as long as a single slice of bread needs to be toasted. For the food processor, the virtual settings might be valid only for the next work unit, and reset to default once the unit is finished or a certain length of time has passed. This functionality would eliminate possible confusion or malfunction, but it would also prevent the virtual functionality from meaningfully supporting or programming the physical functionality.

**Option 2:** An object would always start to operate in its default functionality.
Once a user points the Reality Editor at the connected object, it authenticates its connection and initiates the functionality that the user has programmed with that object. As long as the user’s Reality Editor instance is open and available to the object, the connected object would operate as the present user has programmed it to. The problem with this option is that there are too many possibilities for errors. What if two users want to operate the same object in a similar space? One person could change the functionality without notifying the other, leading, again, to confusion and misunderstandings. Furthermore, it is inconvenient to point the Reality Editor at the object every time it is used.

**Option 3:** A connected object discovers a user the moment the user touches any UI element of the object.
When a connected object identifies the user, it will perform as programmed by that user. This functionality would allow multiple users to operate the same physical objects as long as the interfacing action and the resulting consequence occur within the same object. However, it makes it much more difficult to explain the affordances of objects to others verbally and eventually conflicts may arise out of inconsistent physical functionality. These conflicts would multiply if the interfacing action and the resulting consequences span more than one physical object.

Option 4: The virtual user interfaces follow the model of the physical user interfaces. This means that similarly to the physical user interface, a setting is static. However, since a virtual interface can be changed, the interface becomes temporarily non-static when a user points the Reality Editor at the object and manipulates the settings. From a perspective of consistency, this model provides the highest degree of functionality. Once the user interface affordances become static, a user can communicate them with other users, who can then operate the interface with the new affordances.

Following Don Norman, however, the virtual settings for a machine as described in option 4 might not be directly clear to a user, which can lead to malfunctions. A user therefore would need to be aware of the capability of an object and be able to follow the virtual setup of an object easily. How can one communicate that a static affordance has been made non-static, received a change, and that the change has been defined as the new static affordance of the connected object?

Another question would be whether this statement would be true at all times. There might be scenarios in which it would be more relevant to inform users of the affordance change. For example, a system that provides quick visual feedback with limited physical consequences might not need any communication outside of the feedback loop itself, like a light switch controlling a set of lights. If the chain of action from the light switch to the lights changes, a user realizes quickly which lights are actually controlled by the switch. They are fully self-evident and new affordances can be learned without serious consequences. The same is true of the toaster described above. If a user burns a pattern into the toast, the maximum consequence would be an
unexpected picture on the toast. Not only will the user quickly learn about the change, but it could even be a playful experience or a messaging system among family members.

However, it is a completely different scenario with a food processor. If a user is not aware of the correct operation of the food processor, severe injuries could result. For such a system, a completely different set of interaction needs to be developed. If connected objects find their use in industrial applications or environments where the live of humans depend on the full functionality of a machine, even higher measures need to be considered. Therefore, it makes sense to define categories of use with connected objects.

**Category GREEN**
The green category is used for all affordances that result in no danger from human error. For example, the drawing on a piece of toast can only create incorrect feedback, not harm to a human being. The connected object may only indicate via a small note in the augmented user interface who made the last changes to the object. The new static functionality can be communicated verbally or by direct feedback from the object itself. This category includes most consumer electronics.

**Category YELLOW**
The yellow category is used for all affordances that can result in potentially severe injuries, such as the motor of a food processor. If a user makes changes to an affordance or the linked connections of a YELLOW connected object, all users need to be informed via a messaging systems. The connected object has to continue indicating that a change has occurred for a defined period of time. To ensure that users become aware of the changes, the connected object might not power up until a user has learned via the Reality Editor about the most recent changes. Bluetooth LE could be used for proximity and authentication. Connected objects in the YELLOW category might need to be designed in such a way that human error will be prevented. For example, the object could indicate that there are external connected objects involved in the chain of operations. This category involves objects like kitchen appliances, washing machines, power tools, and so on.
Category RED

The red category is used for all affordances that can result in potentially fatal injuries. For example, assembly manufacturing robots provide functionality that falls into the RED category. If an object or a chain of interaction includes an object in the RED category, only one user at a time should have the right to make changes to the system. Every set of changes needs to be verified with a password, fingerprint, or signature. Another user should only be allowed to operate the changed affordance after the main user has granted permission and the user has learned about the changes. This ensures that human-to-human communication about the changes has occurred. For all other users, the affordances remain static at all time. If a user other than the main user needs to make changes to the system, the main user has to authorize all of the expanded permissions for the new user, and all of the users interacting with the chain of connected objects have to be informed about the change in permissions. As such, a single human or entity maintains responsibility for all changes at any given time. This category involves heavy garden tools like lawn mowers, heavy machinery like assembly robots and CNC machines, and safety-relevant functionalities of cars and other vehicles.

To enable all of these categories an object would need to keep a change log of all of the changes made to its static functionalities. The green category would only need to log the last change, the yellow category would need to keep a private change log, and the red category would require a changelog that is available to all of the people who are working with the machine and/or account for its functionality.

Every category has different motivations and requires different considerations when it comes to the design. The green category requires no additional design changes, since the result of an action is the feedback for the action itself and a user can learn by following the feedback loop. The other two categories would need to consider that operational commands might come from a remote location, not from the machine itself. Therefore, the physical object needs to provide warnings and safety mechanisms, and change according to its new virtual functionality.

As such, connected objects are not entirely self-evident and do not provide a pure example of affordances as Gibson saw them. However, with careful consideration of design parameters, user
cases and social interaction, connected objects can fulfill the concept of affordance as Norman reinterpreted it. Once the augmented nature of a connected object is understood, it becomes self-evident.

8.2. Security and Safety Concerns

This thesis has proposed many answers to the security and safety of connected objects. First, there is the security of local encrypted networks, in which a connected object operates. Second, there is the implemented locking functionality described in chapter 5.3. Additionally, when people interact with objects, one has to consider the social contracts that exist in families or communities. The previous discussion tries to provide more understanding about how connected objects can be handled in different social contexts. However, because connected objects provide invisible connections, not everyone in a social context may be aware of all of them. One solution could be a permanent proactive environment that could provide voice assistance when it detects that a person is becoming irritated by objects that have been reprogrammed and appear to be misbehaving. The objects themselves could also provide more context. The drawback to such solutions, however, is that the overhead of implementation outperforms the utility. The same can be said for augmented reality glasses. Many more technological developments will be needed to make such technology fade seamlessly into our vision. It must be so ubiquitous that the user no longer recognizes it, as fish are said not to recognize water. At that point, a head-mounted display could function as an ambient display that could show the connections between all of a user’s objects or warn the user when the functionality of a particular object has changed.

The security aspects of such connected systems are not trivial. At this point, the IoT may be the biggest threat to the Internet [101]. This is particularly important for connected objects that contain entire services, independent from an external service provider that keeps the systems up to date. The technology described in this thesis could potentially lead to millions of devices that do not receive updates because they are not connected to an external network. On the other hand, since the objects are implemented on a minimal Linux distribution with software that only allows operation on a local network, these objects can be less secure than many IoT objects, which are
exposed to the public Internet. Eventually the Reality Editor server may need to be maintained by a central entity that provides updates. Therefore, the Reality Editor server has been envisioned as a single system; only the objects saved in the folder contain active data. This means that the entire server can be secured with access rights and therefore protected from malware. It also allows simple updates for the entire system, since the system is clearly separated into static files and active files within the object folder.

8.3. A Person as a Connected Object

Can a person be a connected object? At this point, many people do carry smartphones with network IP addresses and unique identifiers such as phone numbers. People are already connected to each other, using chat applications to speak every language known to humanity. The question therefore is not whether a human can be a connected object. Rather, the question is whether the language or the communication protocol of connected objects should be extended into the domain of human communication.

There are applications that track a person’s position and send it to friends. If a person is a connected object, this information can also be sent to a circle of trusted objects that can act on that information. For example, a car can start to warm up when its owner approaches. Different people can get automatic access to Airbnb bookings, or to games. Being a connected object only means that sensors on one’s body speak the Reality Editor platform “language,” not that a person becomes a transparent entity. The Reality Editor augmented reality interface is not a system that provides information about human social networks.

Another interesting direction for this question is not to see a person as a single object, but to see the human body as multiple connected objects. We might already have smart watches and smart shoes that could provide augmented reality interfaces; robotic limbs could also profit from augmented reality interfaces and the Reality Editor platform. For example, a person who wears a robotic leg prosthesis could use the Reality Editor to link it to ambient displays that provide information about its energy status.
8.4. Voice Interfaces

As this thesis is being written, a variety of voice operated interfaces are available, such as Apple’s Siri or Amazon’s Alexa. Increasingly, these devices are used to control the home, which makes it relevant to ask how these interfaces compare to augmented reality interfaces such as the Reality Editor. Voice interfaces allow us to communicate commands to a computer very quickly. However, when it comes to specific information, describing the particular entity in a room that you want to interact with can be cumbersome. If the room is known, this is not a problem—for example, the user could specify that a command for the kitchen lights means a subset of lights in the kitchen. However, the definitions themselves can be troublesome. If the user moves a new light into the kitchen, the voice-driven command structure will not immediately work with the new object. Furthermore, it is difficult to describe the objects in a room without touching or pointing at them.

Once a person can combine a pointing gesture with a voice command [102], however, the entire interaction becomes more intuitive. The Reality Editor can therefore be a powerful tool in combination with voice interfaces. Its communication protocol already provides an interesting semantic for voice interaction. For example, a light with the name “desk light” has two nodes, one named “switch,” and the other is named “brightness.” Instead of using the Reality Editor visual interface, a voice interface could build upon the protocol so that a user could say: “Within the desk light, connect the switch with the brightness.” To take it a step further, if the Reality Editor is combined with a voice interface, one could point it at the desk light and say: “Connect this switch with this brightness.” Since the voice interface semantic and the Reality Editor semantic are so closely related, as both describe objects, users should be able to discover many more interesting overlaps.

At its core, the Reality Editor is a building block in a set of user interfaces that can interweave human computer interaction into traditional human communication so seamlessly that becomes impossible to differentiate a computer interaction from a non-computer supported interaction. At that point, the computer fades into the background and provides a better human experience.
8.5. The limitations of the Reality Editor

This thesis provides an approach to better understanding what augmented reality can contribute to computationally enhanced interaction with physical objects. However, only a limited number of objects can be described as enhanced physical objects (or connected objects). These objects need to have a power source, a computer processor, and electronic input and output capabilities. Some of them might even have speakers or screens. However, this limitation excludes ordinary objects such as spoons, wooden tables, or plates. Such objects can certainly be enhanced with computation, but the result often proves to be more of a toy than a useful tool. Some objects, however, make surprising transitions from the ordinary to the computationally enhanced space, such as the bookmark described earlier. As the technology matures, more ordinary objects will be discovered that become quite powerful when combined with the Reality Editor platform.

Technology-wise, the Reality Editor is still in its infancy. As promising as this research has been, it has also shown that another technological development [103] is required before the Reality Editor can gain its full potential. The Reality Editor is at its most powerful when a user needs to handle a specific subset of a large mass of objects. If a user one only has one or two connected objects, the Reality Editor is a gimmick. Its true power becomes evident in scenarios where a user can interact with hundreds or thousands of objects. A car sharing service is a good example to illustrate its potential. Imagine a parking lot full of rentable cars. Instead of picking a general location, perhaps based on a static map of the lot, and then searching for the right model, a user could simply approach a car and rent it by pointing the Reality Editor at it. Because the Reality Editor is specialized for defining selections, it could also be used to heat individual floor tiles, or to make a selection from a ceiling with hundreds of individually addressable lights and connect these selections with a light switch. Such interactions are nearly impossible without a visual spatial interaction.

As the world around us becomes more modular and more programmable, the Reality Editor will be the digital hammer, saw, and screwdriver to configure and define its functionality.
8.6. Multi-User Scenarios

The discussion about the affordances of connected objects already touches on the question of multi-user scenarios. But another important discussion needs to be held about the proposed solutions in this thesis in comparison with systems that support the Reality Editor by personalizing objects and environments for the active user.

For example, if a user operates a button that activates a coffee machine, the button can be programmed to recognize the user who has activated it and execute the program that this user has attached to the button. If another user presses the same button, the button and the coffee machine may execute a different program, because the entire system of actions is personalized to the user who is currently using the connected objects. It sounds like a good idea, because everyone can tailor the system to their own preferences and always get exactly the outcome they want. But the problem with this scenario lies in the detail. If two users are in the same space at the same time, how will such a system react? If the first and second users work together to make coffee, which program will be executed? How will these users be able to communicate the nuances of coffee making to each other?

The physical world that surrounds us is an important part of interpersonal communication [104]. The Reality Editor should help to support this communication and enable better ways of making changes to the environments that we live in, but not at the cost of social interactions.

We live in a world that only knows one state. It is difficult to negotiate the state of this world with others, but because of these human boundaries we form communities and groups that maintain the same knowledge about the same space. When we build technology that allows each person to see the same world differently, we eliminate these social interactions. In a fully personalized world, the users will be unable to talk to each other about the operation of the coffee machine, because neither will be able to understand the other’s personalized programs. If we scale this problem up to a family of six people and an entire house full of connected objects, everyone may be able to have a personalized experience, but no one will be able to talk with other family members about their individual experience of the world.
As an alternative, think of a set of agent systems that would monitor the use of connected objects. Whenever there are two people using the same object, whose programs differ from each other’s, the agent could mediate between them or present different options that other users have implemented. Such a system would be more complex and require more extensive search functions than this thesis can provide. Without such an agent, however, a personalized Reality Editor would create more confusion in a multi-user scenario than when the users themselves are required to communicate the single state of a system.

8.7. AI vs GUI

It is important to differentiate the previously described supporting agent from an artificial intelligence (AI) system that operates spaces and objects for the user. The agent has a well-defined interface that mediates among the informed intentions and decision of different users, while an AI system can be seen as another person who generates intentions and decisions on a user’s behalf.

The Reality Editor was built to empower human users with better user interfaces for computation. As a result, a user can master its environment, generate new ideas and improve his or her surroundings. AI systems, by contrast, learn about the user’s current state, predict what the user may like or dislike, and make decisions based on this user profile. A non-expert user of such an AI system will always require a third party to make changes and improvements to it. The questions thus arise: Users may be served by AI systems, but who are the AI systems serving? What state of reality will the user be locked into? At what point will a user or a local professional service provider be able to make adjustments to this digital world? Additionally, such AI systems will concentrate control over physical matter into single points. The system described in this thesis, however, enables everyone to understand and adjust the technology. Small businesses can become experts in providing customers with individual solutions. Because improvements can be communicated with other users or service providers, the entire state of connected objects will be enhanced by human operators. There is a significant benefit to placing the keys of innovation in
every humans’ hand and not artificially complicating, automating and centralizing the Internet of Things.

8.8. Standardization

Where to go from here? How can the Reality Editor and its implemented system and communication protocol become a standard among industries? The Reality Editor introduces yet another protocol for communication and therefore contributes to the problem itself. But the chosen protocol for communication is not the important part of standardizing a single protocol for connected objects. This thesis has proven that it is possible to build a simple universal protocol that allows existing connected objects to communicate with future connected objects. The universality of this communication comes from a different user understanding of what a connected object is, when looking at it through the Reality Editor. Where before a user could only chose from abstract predefined items in a computer application, the Reality Editor allows the visual breakdown of any connected object into its functional components. The data that allows the visual breakdown is provided via HTML to the user interface. Because the user understands each object individually, the communication among the connected things can be reduced to the simplest universal data format.

The reason for a multitude of communication protocols for connected objects is that the human understanding of such objects is missing, and therefore all possible choices need to be predefined. As a consequence, companies build closed, limited ecosystems that serve a particular function. However once augmented reality contributes to a better understanding of connected physical things, the communication protocols among these things will organically shift to a simpler form. As this happens, there will be more and more overlap in how many of the existing closed ecosystems communicate data, until a single standard emerges.

The intent of this thesis is not to push for a new communication standard, but for a new perspective on user understanding and user interaction for connected objects. We will not fully understand the impact of IoT if we don’t attain a better understanding of connected objects. A
universal communication protocol was a consequence of the improved understanding of the physical world provided by augmented reality. As these technologies spread, others will hopefully come to the same conclusion. This thesis can thus contribute to finding a single standard for how the connected objects in this world can communicate with each other, while allowing humans to retain control.
9. Conclusion

At the beginning of this thesis, the clear goal was to research a tool that would allow the user a better interaction with connected objects—to make connected objects tinkerable. As the examples and workshops discussed in the thesis show, this goal was reached. A programming system called logic crafting was also developed to support the tinkerability. Another interesting aspect of this research was defining what connected objects are, how we will interact with them, and what it would require to make them part of our lives. The search for answers led to technical developments such as directly mapped digital interfaces and bi-directional augmented reality. Additionally, responses to the technical developments led to specific design considerations. A conversation between the philosophy and the implementation led to a clear definition of what a connected object is and therefore what the Reality Editor as a tool must do. Questions also emerged about how users would interact with ubiquitous connected objects whose abilities are no longer static, once everyone can change their behavior. Possible answers were formulated in a discussion, but their implementation would require future research. The thesis has proven that the philosophy of the World Wide Web can be translated into the physical world and into augmented reality technology.

The directly mapped digital interfaces provide an intuitive user interaction with connected objects that can be operated both digitally and physically. As such, the physical and the digital can ideally merge into one seamless experience. However, it is clear that we are still far from such a merged space. We currently have not only the physical and the digital world, but a world filled with isolated electronics and computers. Looking at the objects that surround us, we might ask how electronics can enhance them, or wonder what computation can contribute to our experiences with them, but this thesis does not propose to saturate the world with electronically enhanced objects that are otherwise ordinary, or with computers programmed to make your toothbrush count the number of times you brush your teeth. This thesis proposes another perspective on the fabric that makes up the technological world around us. Electricity was a luxury once; it has since become such a ubiquitous amenity that one can expect it to exist
wherever one goes. Similarly, computers with a processing capacity once only accessible to expensive research facilities are now in every pocket and used to send smiley faces to friends.

It would be more difficult to tell a scientist in 1960 that computer technology will eventually be used by every teenager in the world to post selfies to the Internet than to explain all the scientific advances that have come out of it. Yet, it is this level of saturation that fundamentally changes how people perceive reality. This thesis shows a working prototype of a system that allows every user to master such a ubiquitous connected world with an augmented reality user interface. The outcome of this programmed world can compete with other systems that might try to learn about a user’s behavior and provide readily composed reality.

In some close or distant future, we will reach the point that every light bulb, every chair, every door around us has some means for connection. However, these things do not come into existence by themselves—they need to have a purpose, technology that drives the purpose, and a user interface that communicates the purpose. This thesis describes a tool that helps us to understand the purpose such future connected objects might have. For example, a bookmark has no special reason to be electronically enhanced. But in a connected world containing an interface such as the Reality Editor, which gives a user a simple and understandable way to control connected objects, a bookmark can suddenly be transformed. Enhanced with electronic components and a computer to network with, the bookmark can become a tool that provides the user with comfort and security, because it can lock the door, close the blinds, and dim the lights. Reaching such a saturation point, where a user can employ the Reality Editor to enhance every object, may take a long time. The technology has to be stable enough to build a new layer of consistent reality, in which objects can exist and function for many years. This thesis presents such technology, and the directly mapped user interfaces provide a promising perspective on how industry can adopt the presented technology to simplify the communications between connected objects and move toward a universal protocol that all such objects can use and understand.

This thesis has proven that it is possible to build such a layer with infrastructure that is independent from external services and built on web technology. The directly mapped user
interface has shown that a user-centric architecture can solve problems regarding the communication among connected objects. Because the Reality Editor provides the user with a fundamental understanding of how the components of a object are put together, it allows the user to make a choice for functionality and connection. This allows the creation of a unified communication protocol that allows up- and downward compatibility.

The pattern of technological advancement is such that battery capacity, processing capabilities, and networking capabilities are consistently improving. This thesis answers the question: How will users employ a connected future for the amenities of daily life? The Reality Editor opens a door for the next stage of a user interface that will support our understanding of connected objects and help humans learn how to interact with and master a digital connected world.
10. List of Figures

Figure 1 A screenshot of the Amiga OS 1.2. 1986 captured via emulator software .................. 12
Figure 2 This diagram shows the four different steps of the approach taken for this thesis ........ 14
Figure 3 A collection of IoT mobile app screenshots .............................................................. 27
Figure 4 An illustration that shows dropdown menus in a smartphone on the right side and physical interfaces on the left side. .............................................................................................................. 28
Figure 5 An augmented reality application shows the interaction with two lights that can be controlled with a bi-directional link. This demo shows that at no time does a user need to memorize names in dropdown menus .................................................................................................... 30
Figure 6 An augmented reality application within an iPad. The application shows a capacitive touch sensor and its data augmented onto the sensor. ........................................................................................................... 31
Figure 7 Mixed reality continuum inspired by Paul Milgram [51]. The bi-directional arrow illustrates that augmented reality can influence physical reality and vice versa. ................. 32
Figure 8 An augmented reality application shows a radio that makes use of a bi-directional link between the augmented and physical reality to provide additional functionality .......... 32
Figure 9 A screenshot from the Reality Editor running on an iPhone. A push button and a light are connected with a virtual patch wire. ................................................................................................. 35
Figure 10 Illustration for the separation of an object into all of its components, represented as simple floating-point numbers. ........................................................................................................... 36
Figure 11 A screenshot from the Reality Editor running on an iPhone. A push button and a light are connected with a virtual patch wire. A user swipes with a finger trough the patch wire to disconnect the objects ......................................................................................................................... 37
Figure 12 First webpage by Tim Berners-Lee (1991) displayed with the Reality Editor on an iPhone 5s (2016). ................................................................................................................................. 41
Figure 13 Diagram for the architecture of the Reality Editor server ..................................... 45
Figure 14 Diagram for the architecture of the Reality Editor mobile app ............................... 50
Figure 15 An illustration showing the openGL rendering pipeline ........................................ 52
Figure 16 An illustration showing the WebKit CSS3D transformation pipeline ...................... 52
Figure 17 An illustration showing the WebKit CSS3D rendering pipeline modified to match the openGL pipeline ................................................................. 53
Figure 18 1) Chinese seal 2) square Kufic, 3) QR code, 4) reacTIVision fiducial markers ....... 59
Figure 19 An ornamental alphabet resembling the Latin alphabet with each letter being represented in a 3x7 pixel grid ....................................................................... 61
Figure 20 Visual evaluations of letters for placing a connecting dot between the letters. .......... 63
Figure 21 Algorithm for evaluating the best connection position between two letters. .......... 63

**Figure 22** The HRQR message “Human Readable Quick Response Code,” with visually separated components ........................................................................................................... 64
Figure 23 Visual representation for the reading algorithm ......................................................... 65
Figure 24 Graphs for column and row mean values .................................................................. 66

**Figure 25** Determining orientation and segmenting the ornamental code ............................. 67
Figure 26 Screenshot from a prototype that shows the square detection, the mean value calculation, the matrix detection and the resulting text decoding .............................................. 68
Figure 27 A screen capture of the Reality Editor showing social media buttons ....................... 71
Figure 28 An illustration shows the steps a user undertakes to save an object reference to the memory bar. a) The user taps on the screen. b) The Reality Editor exposes a big virtual pocket. c) The user drags the finger onto the virtual pocket. Instantly a miniaturized screenshot of the video stream is attached to the finger. At the same time, the memory bar becomes visible. d) The user drags the miniaturized screenshot on to the memory bar to save it .............................................................................................................. 74
Figure 29 An illustration shows the steps a user requires to retrieve an object reference to the memory bar. a) The user taps on to the virtual pocket button. b) The previous step opened the virtual pocket and exposed the memory bar. The user taps a saved memory. c) The Reality Editor shows now a still image of the saved memory. All interfaces associated with objects shown in the memory are fully functional ....................................................... 75
Figure 30 An illustration shows how a user can return from a frozen image back to the normal Reality Editor interface by tapping the activated freeze button ................................................ 76
Figure 31 An illustration shows the steps a user requires to delete an object reference from the memory bar. a) The user taps on the virtual pocket to expose the memory bar. b) The user
taps and holds a memory. After a moment, the memory becomes moveable and a trashcan button is exposed. c) Moving the memory onto the trashcan deletes the memory.............. 76

Figure 32 An illustration showing how a user can interact with visual pointers. a) The Reality Editor shows a memory, indicated by the active freeze button. Tapping on a visual pointer opens another frozen memory (b). c) The Reality Editor shows two objects in a video feed. The user drags a patch wire from one object node to a node on the second object. d) Once the second object is taken out of the camera feed, the patch wire instantly flips to a visual pointer representation..............................................................................................................................................77

Figure 33 An illustration of how a user can connect a visible object with a previously saved object..........................................................................................................................................................................................79

Figure 34 An illustration of how a user can remove a visual pointer (1-2) and save a pointer in the memory bar (2-3). ................................................................................................................................................................................................................................................79

Figure 35 A photograph taken from a computer screen showing the Adobe Edge Animate Software used to generate an Animation for the Reality Editor. ............................................. 81

Figure 36 A screenshot from the Reality Editor that shows a user interface that is currently moveable, as indicated by the visible stripe pattern. .......................................................... 82

Figure 37 A developer uses the Reality Editor developer tools position a web interface .......... 83

Figure 38 A developer uses the Reality Editor developer tools to scale a web interface .......... 83

Figure 39 Screenshot from the Reality Editor Server developer web interface. ......................... 84

Figure 40 Illustrations that show the visual feedback on the Reality Editor while an instant connection is created.................................................................................................................................87

Figure 41 The Reality Editor is pointed at the armrest of a chair. The armrest exposes data points that represent the chair’s functionality.................................................................................. 88

Figure 42 Virtual data points are connected with the data points owned by objects. Each of the robot’s distance sensors is connected to a virtual scaling node connected to a motor. The same distance sensors are also connected to an inverter node that is connected to a motor on the opposite side. The scaling nodes let the robot drive forward. However, when it hits a wall, the inverter sensor becomes dominant and slows down the opposite motor. As a consequence, the robot moves away from the wall. This demo is possible because it processed the state of the entire robot at once. ........................................................................................................91
Figure 43 A screenshot of The Secret of Monkey Island showing the game world, a command panel, and the player's inventory. The user can store found items in this inventory. .......... 92
Figure 44 Screenshot from the Minecraft crafting table.......................................................... 93
Figure 45 This logic crafting interface appears when the user taps on a logic node. .............. 94
Figure 46 When the user taps on the green virtual pocket button, a block selection menu appears. .......................................................... 95
Figure 47 When the user places a finger on a block and starts moving it, the block attaches to the finger and the block selection menu disappears so that the user can place the block on the logic crafting board. The block is red when it can not be placed.................................. 95
Figure 48 If the user places the block on a free grid position, the block becomes green. This color indicates that the block can be placed at this position. .................................................. 96
Figure 49 A logic block with a size of three blocks is placed onto the logic crafting board....... 96
Figure 50 Screenshot from a delay switch logic crafting program. In the present version, inputs and outputs are labeled with IN and OUT. A later iteration could have custom names..... 103
Figure 51 Screenshot from a delay cascade logic crafting program................................. 103
Figure 52 Screenshot from a seesaw switch logic crafting program............................ 104
Figure 53 Screenshot from an or logic crafting program.................................................. 105
Figure 54 Drawings for logic blocks that would snap automatically into each other......... 106
Figure 55 Different designs for how a logic node appears when connected..................... 109
Figure 56 Draw a line to connect to nodes................................................................. 110
Figure 57 Swipe through a line to disconnect the nodes.................................................. 110
Figure 58 Hold and move to reposition a node.............................................................. 111
Figure 59 Left: The normal virtual pocket button. Right: The variation used in the logic crafting board. ................................................................. 111
Figure 60 A screenshot from the Reality Editor showing the settings for a threshold logic block Setting menus can be customized with HTML and JavaScript by designers or developers who build logic blocks................................................................. 112
Figure 61 A screenshot from the settings menu for the logic node. The overall logic node setting allows a user to define names for each color input and output. It also allows the user to select an image as icon to be placed in the logic node. A switch can set the logic node to open the last block settings menu, for when a user taps a logic node. This allows a user to
make changes to the behavior of the logic node, without being exposed to the logic crafting board. .......................................................... 113

Figure 62 Once the Logic Node becomes moveable, the sidebar changes its functionality to allow the user to either trash or save the node. .......................................................... 114

Figure 63 A screenshot from the Reality Editor showing selectable user interfaces. The interfaces are animated in the Reality Editor so that they reveal their capabilities to the user. .... 115

Figure 64 Illustration for AR push buttons. .......................................................... 116

Figure 65 Illustration for different form factors of AR sliders. ........................... 116

Figure 66 Illustration for AR kinetic sliders. .......................................................... 117

Figure 67 Illustration for AR data visualizers. .......................................................... 118

Figure 68 Illustration for AR sliders with feedback. ................................................ 119

Figure 69 Illustration for 6-DoF kinetic sliders. .................................................. 120

Figure 70 The color scheme that is used throughout the entire Reality Editor user interface.... 122

Figure 71 A program that modeled with physical and virtual objects (blocks).................. 124

Figure 72 A screenshot from the Reality Editor showing an early node-based programming interface on a robot. .................................................................................. 125

Figure 73 User interface components augmented on to a Lego robot. ....................... 126

Figure 74 The Reality Editor showing a smart grid application. ............................... 127

Figure 75 A screenshot from an augmented reality experiment that shows a temporary tattoo used as a marker.................................................................................. 128

Figure 76 A screenshot from the Reality Editor showing artwork augmented with web services. .................................................................................. 129

Figure 77 A screenshot of the Reality Editor showing the interaction with a general-purpose knob and a light.................................................................................. 130

Figure 78 A screenshot of the Reality Editor showing two Arduino Yun boards being connected. .................................................................................. 131

Figure 79 A user interacts with a connected object light............................................. 133

Figure 80 A illustration for the car scenario user interaction.................................... 133

Figure 81 Four nodes that represent windows in a car are daisy-chained so that they execute the same signal input.................................................................................. 134

Figure 82 A user operates car windows remotely...................................................... 135
Figure 83 With the crafting mode, a user connects a physical rotation knob to two nodes that are placed on a marker. Each of these nodes represents one of the digital properties of a teapot.

Figure 84 Once the connection from Figure 83 is created, the user switches to the object interaction mode where a 3D teapot is visible. Operating the physical rotation knob changes the shape of the teapot.

Figure 85 A robot is operated via an augmented reality bi-directional feedback loop.

Figure 86 OS X standalone application for serving the Reality Editor mobile app.

Figure 87 A screenshot taken from the Reality Editor interface that shows two Lego WeDo 2.0 blocks with superimposed directly mapped nodes for all attached and built in components. Beside a built-in button and LED light, each Lego WeDo 2.0 block has a distance sensor and a motor attached. The shown program connects a button to a logic node and a LED light. The logic node is connected to a second LED light. The logic node is programmed to invert the signal, so that when the button is pushed one LED light starts glowing while the other stays dark and vice versa for when the button is not pushed.

Figure 88 Screenshot from a secure payment transaction.

Figure 89 A user selects personal preferences for the spatial search.

Figure 90 The Reality Editor displays spatial search results.

Figure 91 A picture taken at the RCA workshop.

Figure 92 Connected bookmark by Fiona O'Leary. The bookmark can sense whether a book is closed.

Figure 93 Augmented Reality Pill Box by Václav Mlynář. The pill box can sense when new pills need to be ordered and provide the user with options for purchasing them.

Figure 94 Augmented Reality Card Key by Kyungmin Han. The user can see floor access by pointing the phone at the key card.

Figure 95 The user points a Reality Editor at objects that have an ornamental code sticker attached. Object-related information is displayed.

Figure 96 A toaster and a food processor.

Figure 97 A screenshot from the Amiga OS Desktop—Graphical User Interface.

Figure 98 A toaster and a food processor with augmented user interfaces forming connected objects.
11. Reference


[17] K. Č. Pucić and P. Coulton, “[Poster] Utilizing contact-view as an augmented reality


[35] “ninja blocks.”


Kodak, There’s a photographer in your town. 1913.

(openframeworks.)


“Vuforia.”


W. DENSO, “QR code. ISO/IEC18004, June 2000.”.


[83] “Pure Data.”


[85] “Arduino.”


[90] “Reality Editor source code.”


[94] “discourse forum.”


12. Appendix

12.1. AR User Interface Reference API

Include the open hybrid functionality by instancing as following in your Java Script code:

```javascript
var obj = new HybridObject();
```

**.addReadListener([IO Point], callback)**

Every communication with the object is happening passively. This means that the object is not pushing data. If a user interface wants to read data from a Hybrid Object it first needs to send a read request.

Example:

```javascript
obj.addReadListener("led", function(e){
    input = e*255;
});
```

**.write([IO Point], [Value 0 ~ 1])**

You can write to the Hybrid Object with write(). The scale of your values should be between 0.0 and 1.0.

Example:

```javascript
obj.write("led", output);
```

**.sendGlobalMessage([message])**

Send broadcast messages to all other objects currently visible in the Reality Editor.

Example:

```javascript
obj.sendGlobalMessage("Hello World");
```

**.addGlobalMessageListener(callback[message])**
 Allows you to listen to messages send to all other objects currently visible in the Reality Editor.

Example:
obj.addGlobalMessageListener(function(e){
    console.log(e);
});

.subscribeToMatrix()
Forces the Reality Editor to send the 3D transformation Matrices.

Example:
obj.sendGlobalMessage();

.addMatrixListener(callback[modelViewMatrix][projectionMatrix])
Allows you to listen to updates 3D-transformations. It only works once subscribeToMatrix() has been called. This listener is synchronized with the video update rate.

Example:
obj.addMatrixListener(function(e,f){
    modelview = e;
    projection = f;
});

.setFullScreenOn()
Forces the Reality Editor to show your content full screen without 3D transformation. This comes in handy for when you want to use the transformation matrices directly.

Example:
obj.setFullScreenOn();

.setFullScreenOff()
Does the oposit from setFullScreenOn.

Example:
obj.setFullScreenOff();

.addVisibilityListener(callback[e])
Allows you read if the interface is visible or not. The interface stays active for 3 seconds after it becomes invisible.

Example:
obj.addMatrixListener(function(e){
    visible = e;
});

.getPositionX(), getPositionY(), getPositionZ()
Returns a number for translation distance and position between the iOS device and the marker.

Example:
x =  obj.getPositionX();
y =  obj.getPositionY();
z =  obj.getPositionZ();

12.2. Hardware Interface Reference API
This API is intended for users who want to create their own hardware interfaces. Hardware interfaces are adapters that let you interact with different kinds of objects, like Arduino Uno or Lego WeDo, or even a virtual timer.

.enableDeveloperUI(developer)
Enables the developer mode for all HybridObjects and enables the developer web interface.

Example:
server.enableDeveloperUI(true);

**addNode(objectName, nodeName, type)**
Adds a new IO point to the specified HybridObject.

Example:
server.addNode("lego1", "light1", "node");

**renameNode(objectName, oldNodeName, newNodeName)**
Renames a node in the specified HybridObject.

Example:
server.renameNode("lego1", "light1", "motor1");

**activate(objectName)**
Activates a specified HybridObject when it is done being configured with new nodes.

Example:
server.activate("lego1");

**deactivate(objectName)**
Deactivates a HybridObject so its data is no longer processed by the server until it is activated again.

Example:
server.deactivate("lego1");

**write(objectName, nodeName, value, mode, unit, unitMin, unitMax)**
This function writes the values passed from the hardware interface to the HybridObjects server.
Mode specifies the datatype of value, you can define it to be whatever you want. For example 'f' could mean value is a floating point variable.

Mode, unit, unitMin, and unitMax are currently optional, and will respectively take on default values of 'f', false, 0, and 1.

Example:
server.write("lego1", "light1", Math.random(), "f");

**addReadListener(objectName, nodeName, callBack)**
This function attaches a callback to a HybridObject's node to listen for new values it receives and process them accordingly.

For example, the arduino hardwareInterface listens for values and writes them to the serial port.

Example:
server.addReadListener("lego1", "light1", function (data) {
    console.log(data.value);
});

**removeReadListeners(objectName)**
Removes the callbacks attached to a HybridObject by addReadListener.

Example:
server.removeReadListeners("lego1");

**addEventListener(option, callBack)**
Listens for specific events from the server. Option can be either "reset" or "shutdown."

Example:
server.addEventListener('shutdown', function() {
console.log('shutting down');
});

map(x, in_min, in_max, out_min, out_max)
Remaps the value of x from the range of \([\text{in\_min}, \text{in\_max}]\) to the range of \([\text{out\_min}, \text{out\_max}]\)

Example:
var scaledValue = server.map(data.value, 0, 1, -100, 100);

advertiseConnection(object, node, logic)
Broadcasts the HybridObject and node names to the Reality Editor for the Instant Connection feature.

Example:
server.advertiseConnection("lego1","light1");

getDebug()
Checks if debug mode is turned on. Returns true if debug mode is on, false otherwise.

Example:
if (server.getDebug()) console.log("developer");

12.3. Arduino Reference API

In the Arduino code the Hybrid Objects library needs to be instantiated as followed:
#include <HybridObject.h>
HybridObject obj;

.developer()
If called, the developer() function allows you to access all developer functionality. It allows you

to move and scale user interfaces within the Reality Editor and it gives you access to the
Example:
void setup() {
    obj.developer();
}

.add([Hybrid Object], [IO Point])
With the .add() function you can add new IO Points to your Hybrid Object. IO Points can be input, output or both. The direction is not relevant. The added IO Points will be represented in the Reality Editor.

Example:
void setup() {
    obj.developer();
    obj.add("object", "led");
    obj.add("object", "sensor");
}

.read([Hybrid Object], [IO Point (String)])
Within the Arduino loop() function you can use .read to read data from an IO Point. All values are floating point in the range between 0 and 1.

Example:
void setup() {
    obj.developer();
    obj.add("myObject", "led");
}
void loop() {
    int input = obj.read("myObject", "led");
}

.write([Hybrid Object], [IO Point], [Value 0 ~ 1])
Within the Arduino loop() function you can use write to an IO Point. All values are floating point in the range between 0 and 1.

Example:
```c
void setup() {
    obj.developer();
    obj.add("myObject", "led");
}
void loop() {
    int output = 123;
    obj.write("myObject", "sensor", output);
}
```

.map([Value], [Value Min], [Value Max])
Since all data is floating point between 0 and 1, the map function scales your data into this scale. For example if you have a range between 0 and 100 and your value is 50 map() will return 0.5.

Example:
```c
void setup() {
}
void loop() {
    int input = 123;
    float output = obj.map(input, 0,200);
}
```