An Augmented Reality Editor: Building data-focused tools to extend the capability, connectivity, and usability of a mobile Internet of Things browser

by Benjamin F. Reynolds

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

May 2017 [June 2017]

© 2017 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: 
Benjamin F. Reynolds
Department of Electrical Engineering and Computer Science
May 25, 2017

Certified by: 
Pattie Maes
Professor of Media Arts and Sciences
Thesis Supervisor
May 25, 2017

Accepted by: 
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

[Signature]
An Augmented Reality Editor: 
Building data-focused tools to extend the capability, connectivity, and usability of a mobile Internet of Things browser

by Benjamin F. Reynolds

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology

May 2017

© 2017 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Signature redacted

Department of Electrical Engineering and Computer Science
May 25, 2017

Certified by:

Signature redacted

Pattie Maes
Professor of Media Arts and Sciences
Thesis Supervisor
May 25, 2017

Accepted by:

Signature redacted

Christopher Terman
Chairman, Masters of Engineering Thesis Committee
DISCLAIMER NOTICE

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

The images contained in this document are of the best quality available.
An Augmented Reality Editor: Building data-focused tools to extend the capability, connectivity, and usability of a mobile Internet of Things browser

Benjamin Reynolds

Submitted to the Department of Electrical Engineering and Computer Science on May 26th, 2017, in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

Abstract

This thesis describes extensions made to the Reality Editor system to improve its ability to browse augmented reality content and to customize the flow of data between Internet of Things objects. The augmented reality browsing is improved by a platform shift to web technologies, while the data customization is provided by the design of logical operators that can be added to a network of objects.

The contribution lies in the implementation of a novel grid-constrained, data flow visual programming language called Logic Crafting, which provides a modularized and extensible environment for manipulating the data passed between IoT objects. Associated contributions involve web service modules that allow non-developers to connect IoT objects with arbitrary remote services, and an implemented mechanism for changing the permissions model for IoT objects on a per-resource level. The thesis then presents a number of prototypes enabled by using web technologies for augmented reality, as well as demonstrative use cases for programming IoT objects with Logic Crafting. Finally, the thesis reflects on the implication of these features and prototypes on the subject of augmented multiuser environments.

Thesis Supervisor:

Prof. Pattie Maes
Professor of Media Arts and Sciences
# Table of Contents

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

## 1. Introduction

1.1. Motivation

1.2. Contributions Overview

## 2. Background and Related Work

2.1. Augmented Reality

2.2. The Web of Things

2.3. Data Flow Visual Programming Languages

2.4. Persistent Data Systems

2.5. Secure Data Sharing

## 3. Research Question

3.1. Author AR content and place it in the world

3.2. Allow AR content to affect and be affected by the physical world

3.3. Customize networks of smart objects with emergent behavior

3.4. Collaborate to edit, share, and remix deployed creations

3.5. Control one’s data privacy and provide adequate security

## 4. Augmented Reality Content Authoring and Browsing

4.1. AR Browsing (iOS Web View Bridge)

4.1.1. Integration

4.1.2. Implementation

4.2. AR Authoring and Deployment

4.2.1. Hybrid Object API

4.2.2. Unconstrained Interface Positioning

4.3. Discussion

## 5. Logic Crafting

5.1. Design and User Flow

5.1.1. Node Interactions

5.1.2. Logic Crafting Interactions

5.2. Features and Implementation

5.2.1. Front-End Implementation
5.2.1.1. Data Structures 40
5.2.1.2. DOM Structure 43
5.2.1.3. Routing 43
5.2.1.4. Touch Interaction 49
5.2.1.5. Block Menu 51
5.2.1.6. Design Enhancements 53
5.2.2. Back-End Implementation 55
5.2.2.1. Realtime State Synchronization 55
5.2.2.2. Invisible Node Interface 58
5.2.2.3. Logic Blocks 60
5.2.3. Web Service Implementation 63
5.2.3.1. WebPost 65
5.2.3.2. WebListen 66
5.2.3.3. TriggerIFTTT 67
5.2.4. Locking Implementation 68
6. Prototypes 72
6.1. Augmented Reality Content Authoring and Browsing 72
6.1.1. Augmented Art Gallery 73
6.1.2. Immersive 3D AR 74
6.1.3. Context-aware AR 76
6.1.4. Back-end connected AR 77
6.1.5. Menu-reactive AR 78
6.1.6. Industry-scalable Spatial Search 81
6.2. Logic Crafting 82
6.2.1. Lego Playground 82
6.2.1.1. Start and Stop 84
6.2.1.2. Delay Chain 84
6.2.1.3. Arithmetic Function 85
6.2.1.4. If Both Are On (Logical AND) 85
6.2.1.5. If Either Are On (Logical OR) 86
6.2.1.6. Rotating Car 87
6.2.1.7. Engine On/Off Switch 87
List of Figures

Figure No.  
Connecting objects with the Reality Editor  
System diagram for embedded Web View  
Unconstrained repositioning  
Pseudocode for drawPlaneIntersection  
Connecting a logic node  
Crafting view  
Logic Crafting visual specification  
Logic Crafting data structures  
Pseudocode for valid paths  
Rendered valid paths  
Pseudocode for legible paths, step one  
Rendered legible paths, step one  
Pseudocode for legible paths, step two  
Rendered legible paths, step two
15 Link rendering results
16 Touch interaction state machine
17 Design enhancements comparison
18 Visual feedback for block repositioning
19 Crafting board node preview
20 Invisible links server architecture
21 Implemented blocks
22 JavaScript for seesaw block
23 JavaScript for webPost block
24 Visual feedback for node locking
25 AR web content prototype
26 AR art gallery prototype
27 AR 3D prototype
28 JavaScript for AR 3D prototype
29 Context-aware AR prototype
30 AR live data prototype
31 HTML for menu-reactive AR interfaces
32 Menu-reactive AR prototype
33 Spatial search AR prototype
34 Lego WeDo hardware for logic crafting prototypes
35 Connected Lego car
36 Start and stop crafting board
37 Delay chain crafting board
38 Arithmetic function crafting baord
39 Logical AND crafting board
40 Logical OR crafting board
41 Rotating car crafting board
42 Engine on-off switch crafting board
43 webPost crafting board and server code
44 webListen crafting board and request
45 Online spreadsheet triggerIFTTT prototype
46 Push notification triggerIFTTT prototype
1. Introduction

1.1. Motivation

The Internet of Things is a paradigm in which sensors, actuators, and other objects are augmented with wireless communication technologies such that any IoT object can interact with, and inform the behavior of, any other accessible objects [1]. For over a decade, the IoT has promised to make people's lives easier and more productive, with environments smart enough to assist us with tasks, or to be automated altogether. There is no single standard for IoT technologies, so the majority of currently deployed IoT products use proprietary solutions that do not interoperate with one another, and require separate software applications to configure. As a result of this restricted interoperation and clumsy usability, most IoT-enhanced environments become only marginally smarter than those without IoT objects.

The Reality Editor solves this problem by introducing an open source IoT platform and an accessible tool for connecting objects within it [2]. The Reality Editor is a smartphone application that provides an augmented reality interface for interacting with IoT objects. The system associates a virtual object with each physical IoT object using an open source platform [3] to form "Smarter Objects." These objects provide all the benefits of a tangible user interface, but also provide an auxiliary virtual user interface and are able to communicate with a network of nearby objects [4]. A user can detect Smarter Objects by pointing their phone camera at a unique fiducial marker associated with the object. This interaction reveals a virtual user interface which appears to float above the Smarter Object's position in 3D space on the phone screen. Manipulating the virtual user interface sends commands to control the physical object, and vice versa.

Figure 1. Connecting objects with the Reality Editor interface [2].
Beyond augmenting physical objects with virtual user interfaces, the Reality Editor also provides a way to intuitively connect objects into a flow-based network, as seen in Figure 1. Objects are decomposed into Input/Output (IO) nodes, which each hold the value of a single sensor or actuator within an object. For example, each light bulb in a lamp would have an IO node with a floating point value between 0 and 1 representing the brightness. Each knob on a radio would have an IO node with a floating point value between 0 and 1 representing the current rotation. Data from any IO node can be streamed into any other IO node, creating realtime causal relationships among Smarter Objects' behaviors. For example, if a radio knob is visually connected to a lightbulb's IO node within the Reality Editor, changes to the knob are then transmitted to the bulb, which can process this data and update its physical and virtual state accordingly. This is useful because Smarter Objects can be configured to behave in ways that assist the user even when the Reality Editor app is put away. For example, a user can customize the light color of their lamp, and connect it to a sensor on their chair that turns the lamp on or off depending on the user's presence. In this manner, the Reality Editor is used to construct smart environments around the user.

However, the Reality Editor is limited in how "smart" of an environment a user can construct. As it stands, the Reality Editor can relay data between objects, which immediately sends the state of A to B. This is sufficient to use an object as an on-off trigger, for example "turn on the radio when the user opens the door," or to match another object's analog value, such as "set the volume of the radio to the brightness of the lamp." But more complex scenarios cannot be built with the Reality Editor. For example, a light can't learn what brightness a user prefers and turn on with that value when the car pulls into the driveway. A radio can't pause only when all lights in the house have been turned off. An irrigation system can't water its plants only between certain hours and if a light sensor's value exceeds a certain threshold. More generally, the system cannot monitor patterns of object usage over time and alert users or objects when something goes wrong, or to log objects' data in the cloud for future analysis. At best, some of this logic could be programmed into the hardware, which could expose
additional IO nodes for different scenarios that it is predefined to understand, but some of these situations would be entirely infeasible with the existing Reality Editor system.

Another problem is that the Reality Editor can only connect Smarter Objects to other objects within the same private local network. This is sufficient to construct a smart home environment, but falls short as soon as one wishes to connect their car to objects within their home, which likely connect to a different network. It similarly fails to automate office or industrial environments that exist across large campuses or distant geographic locations. And there is no way for a Smart Object to connect to another important type of smart “objects”: applications running on the Web. Web services can provide specialized computations that build upon the symbiotic ecosystem of the Web and the actions of a huge number of connected clients. Being able to connect Smarter Objects to these would enhance their possible behavior, but it is impossible to do so in the existing Reality Editor.

Beyond IoT devices, the Reality Editor also makes a user's environment smarter by augmenting physical spaces with AR interfaces containing useful information or interactive controls. But the process for creating these interfaces was laborious and required expertise with C++, so the number of deployed interfaces remained low. To make using the Reality Editor worthwhile on a frequent basis, it is necessary that there is at least a certain amount of content that can be consumed.

In this thesis, I explore solutions to the aforementioned limitations of the Reality Editor, by extending it with additional functionality. There are two key areas that the previous iteration of the Reality Editor needs to improve: it needs more content to view and objects to connect, and it needs to do more with the resources it encounters; for interfaces this means enabling more meaningful interactions, and for objects this means enabling more intelligent behavior with its connections. Thus, the work of this thesis aims to improve the connectivity of the Reality Editor and the capability of the objects it connects, by adding features that make better use of the data flowing between objects. All of this must be accomplished without detracting from the usability of the system. It
can only make users' lives easier if the technology is accessible and free of frustration. The motivation for these improvements can be summarized by the following three questions, in order of least to most importance:

- How easily can a user connect and configure objects? *(Usability)*
- How many interfaces can be viewed, objects connected, and of what variety? *(Connectivity)*
- Once objects are connected, what can they do that they could not before? *(Capability)*

The deeper motivation for these improvements are the following research questions:

- How can we build tools to better allow people to control and interact with IoT objects in a usable and meaningful way that improves their lives?
- How can we build tools to better allow people to easily prototype, publish, and browse meaningful AR content?

### 1.2. Contributions Overview

The goal of our project was to take an already powerful tool, the Reality Editor, and extend it by building up a comprehensive ecosystem for building, deploying, browsing, and customizing bidirectional AR content. Such a system should provide the capabilities to easily and intuitively:

- Author AR content and place it in the world
- Allow AR content to affect and be affected by the physical world
- Customize networks of smart objects with intelligent, emergent behavior
- Collaborate with other users to edit, share, and remix deployed creations
- Control one's data privacy and provide adequate security for intended applications

Our work has addressed each of these points with a number of enhancements and additions to the Reality Editor app and server, as outlined in Table 1. This work was accomplished by a team of three researchers, each with a specific focus and responsibilities. The solutions I directly worked on are shown in boldface:
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Author AR content and place it in the world | • Support for web content and associated authoring tools  
• Direct manipulation interface for spatial positioning |
| Allow AR content to affect and be affected by the physical world | • IO nodes for reading and writing data to hardware  
• Hardware Interfaces act as adapter between software and each type of hardware  
• API for updating AR interface with IO data |
| Customize networks of smart objects with intelligent, emergent behavior | • Logic nodes provide a flow-based programming environment for adding logic between IO nodes  
• Ability to connect to web services |
| Collaborate with other users to edit, share, and remix deployed creations | • Realtime synchronization between concurrent users editing the system |
| Control one’s data privacy and provide adequate security for intended applications | • Data is distributed in local network rather than passing through a central authority  
• Interfaces use web-standard security protocols  
• Permission control system allowing users to lock access to sensitive resources |

Table 1. Desired capabilities (left) and the implemented solutions (right). The author's contributions are displayed in boldface.

I also contributed to the project by building a variety of prototypes that tested different aspects of the system. The results serve as inspiring use cases for the potential of AR and IoT. The demos I built are divided into a number of categories. Related to AR content authoring and browsing are a number of prototypes that show the capabilities of the Hybrid Object JavaScript API. These include demos that visualize live data to customers at a grocery store, and a demos that display immersive multimedia content to the user. Related to extending the capability of connected IoT objects are my prototypes that use Lego WeDo sets as a playground for inserting logical operators and small programs into the network to affect its behavior. I also prototyped interactions that connect with web services to achieve functionality outside of the limits of the local network of connected IoT objects.
2. Background and Related Work

Although the prevalence of the Internet of Things, augmented reality, and visual programming environments accessible from mobile devices are relatively recent phenomena, each contribution of this thesis builds on decades of research. Chapter 2.1 discusses the history of AR and the current development landscape for AR applications. Chapter 2.2 discusses the Web of Things and related communication protocols, which relates to the integration of web services into networks of Smarter Objects. Chapter 2.3 discusses existing data flow visual programming languages which serve as inspiration for the Logic Crafting system. The related field of persistent data management is explored in Chapter 2.4, which did not lead to any specific features in this thesis but serves as a background that influenced the design of Logic Crafting and web service integrations in the Reality Editor. Inspirations for permission-based resource sharing are discussed in Chapter 2.5.

2.1. Augmented Reality

Augmented Reality (AR) allows users to see the real world superimposed with virtual objects via an optical or video display. More formally, AR is defined by Azuma in "A Survey of Augmented Reality" [5] to be any system that "combines real and virtual", "is interactive in real time", and "is registered in three dimensions." The first such display was created by Sutherland in the 1968 [6], in fulfillment of his imagined "ultimate display" [7] that would take the form of "a room within which the computer can control the existence of matter." Many seminal ideas and prototypes were introduced in the early 1990s, leading to the first AR conference, IWAR 98, in 1998 [8]. The explored application domains include medical, manufacturing and repair, annotation and visualization, robot path planning, entertainment, and military.

In the past, the most limiting problem for wide-scale AR systems has been registering and tracking the physical surfaces to augment. Computer vision advances in the 2000s, however, enabled "augmented ubiquitous computing" [9], which makes use of sufficiently powerful mobile devices to interact with AR content. More recently, improved
processing power along with improved tracking techniques (including magnetic, vision-based, inertial, and GPS methods), have brought AR to the brink of mainstream adoption, with commercial AR tools, systems, and browsers readily available. Some of these include Wikitude and Layar, which also provide standalone authoring tools for AR content, as well as SDKs such as Vuforia and ARToolkit that allow developers to integrate tracking into existing applications [10]. While these tools and platforms are robust, they do not address what Heun calls "bidirectional augmented reality" [11] in which the physical and the virtual can interact in both ways; virtual interfaces can affect physical objects in addition to the traditional model of AR. Bidirectional AR is less thoroughly explored, and tools and browsers for such content are not as prevalent as those for general AR.

2.2. The Web of Things

Providing an interface for communicating with web services builds on the idea of the Web of Things (WoT), in which smart objects are treated as first-class citizens of the Web that support HTTP communication and are able to interact with other web services and applications in a loosely-coupled system [12]. Several technologies and architectures exist to facilitate loosely-coupled interactions among WoT objects and web services. The first is the RESTful API, in which URIs define an interface to which clients can send requests to interact with services. However, this is a client-pull model, which does not fit the model of a Smarter Object system in which streams of data should be continuously pushed to their destinations. A relevant WoT design that uses a pushing infrastructure includes a WebSocket and a Webhook in each object [13]. The objects subscribe to frequent events by opening a WebSocket connection, and respond to infrequent events with a Webhook. Understanding these approaches is important when designing a communication method between the Reality Editor and web services.

2.3. Data Flow Visual Programming Languages

When designing a visual programming language (VPL), the data flow model is a frequent choice, and if often used in the domains of music, constructing user interfaces,
image processing, and science [14]. Examples include Pure Data (music) [15], Fabrik (user interfaces) [16], VIVA (image processing) [17], and LabVIEW (science) [18]. These languages are influenced by VPLs as well as data flow languages, which are not always the same. Many recent VPLs are influenced by Scratch [19] and Blockly [20], which are block-based languages intended for playful and educational multimedia purposes. While they use a procedural rather than a data flow paradigm, they offer inspiration about designing an accessible user interface. In the space of IoT, the Distributed Data Flow model [21] processes data between connected objects, making a distinction between IO Nodes and Compute Nodes, but doesn't provide a visual programming language for inserting logic on the application layer. The state of the art in visual programming languages as they relate to the Internet of Things is presented by Ray in "A Survey on Visual Programming Languages in Internet of Things," [22] in which more than a dozen such languages are compared, including Node-Red, NETLab Toolkit, Scratch for Arduino, DGLux5, and AT&T Flow Designer. The author concludes that the benefits of these systems include the natural fit with IoT interactions, reduction of "syntax errors," and support for touchscreen user interactions. Common problems identified include "poor user interface, slow code generation, lack of standardized model, and absence of abstraction layer." Future data flow VPLs should be careful to address those problems.

2.4. Persistent Data Systems

In most situations, users who interact with persistent data in Database Management Systems are trained programmers and engineers who can be expected to write complex SQL statements to filter and process data. This is a problem for integrating data management tools into the Reality Editor, as most users have no engineering background and cannot be expected to query data with SQL. Visual Query Languages are a substitute for SQL that use graphical representations to perform queries [23]. Most frequently, this takes the form of a directed graph diagram describing the formal relations of the query, such as in the SmartVortex Visual Query System (VQS) [24], which is designed for working with data streams in industrial engineering applications. This system associates a graphical node with each SQL keyword. Connecting blocks'
inputs and outputs constructs a query without the need for programming. However, designing a query still requires the same conceptual expertise from the user, which cannot be expected of the Reality Editor’s users.

A more appropriate inspiration is the Aurora system [25], which is a VQS designed for managing data streams for monitoring applications. It filters and processes data in a more intuitive way, by routing input data streams through a visual network of operator boxes that filter, aggregate, and merge streams of data to produce a desired output stream. The system provides more power by allowing the user to store historical data when desired. The Reality Editor can take inspiration from the intuitive flow-based operator boxes, and the power to perform continuous and one-time queries in a single system. However, these ideas need to be even further simplified to integrate into the minimal augmented reality GUI of the Reality Editor. An additional system that applies a visual series of constraints to filter a dataset is described in [26].

To simplify the data management system as much as possible while still remaining useful, the Reality Editor could exploit the time-ordered, sequential property of its data. Users may only be concerned with recent data, or may be looking for data trends over a periodic interval. Temporal Visual Query Languages (TVQLs) have been designed specifically to navigate time-related data [27], and often use dynamic query sliders [28] or brushing histograms [29], as visual interfaces for filtering data. The Reality Editor may benefit from a simple tool inspired by these examples that allows users to access data from relevant time intervals. To understand periodic data, the Reality Editor may benefit from the visualizations presented in [30] and [31]. The primary relevance of these data management systems is for illustrating how complicated a user interface must be to provide complete control over a system’s data. Researchers have developed a number of interfaces for simplifying these interactions as much as possible, but even the best is disproportionately complicated compared to the core functionality of the Reality Editor. These limitations must be taken into account when deciding which features to include in the Logic Crafting system, and what functionality should be outsourced to an external web service.
2.5. Secure Data Sharing

Controlling the set of users able to access or modify a certain object within the system builds naturally upon the ideas of access control lists (ACLs) and permission-based security models [32]. ACL systems allow a subject to perform an action on an object only if the subject has been given those permissions, such as in the UNIX file system [33]. UNIX gives each file separate sets of permissions defining what the owner of the file can do with it and what other users can do with it. Permissions include the ability to read from, write to, and execute the file. The Reality Editor can use these ideas to restrict the set of users who can access, modify, or share sensitive objects within the network. UNIX also has the concept of a super-user who can access and change permissions of any file. Users can enter super-user mode to modify secure files, then exit the mode to prevent accidental or malicious changes. This Reality Editor may benefit from a mode similar to super-user, where a user can lock or unlock access to certain objects.
3. Research Question

The goal of this thesis is to take an already powerful tool, the Reality Editor, and extend it by building up a comprehensive ecosystem for building, deploying, browsing, and customizing bidirectional AR content. As previously outlined in Table 1, such a system should provide the capabilities to easily and intuitively:

- Author AR content and place it in the world
- Allow AR content to affect and be affected by the physical world
- Customize networks of smart objects with intelligent, emergent behavior
- Collaborate with other users to edit, share, and remix deployed creations
- Control one's data privacy and provide adequate security for intended applications

The motivation for each of these questions is explained below along with an individual research question that I will attempt to answer in this thesis.

3.1. Author AR content and place it in the world

If we are to live in a world where AR is ubiquitous, then that world needs a lot of AR content in it. One needs to be able to look through a device at any time or place and be rewarded with additional information, controls, or entertainment. Just as the World Wide Web would be nothing without the websites that populate it, an AR browser is nothing without content to interact with. The World Wide Web's content has been built up over decades. This has been helped by the fact that the entire pipeline for creating compelling websites has become easier and more powerful over time through a variety of tools and services, such as usable IDEs and design tools, improved programming languages, a surplus of frameworks, and powerful content management systems. In short, the content creation methods for the Web contribute to its success. In this thesis, I hope to pursue an analogous question for the Reality Editor. How can we improve the content creation process for the Reality Editor to make it easy for developers to deploy meaningful AR experiences into the world?
3.2. Allow AR content to affect and be affected by the physical world

AR is usually thought of as a unidirectional system; it sends data from the physical environment to the user, via a virtual interface. However, to further break down the boundaries between the physical and the virtual, a closed feedback loop is needed. The system must also be able to send data from the user to the physical environment, via the virtual interface. This makes AR a tool that can be used to manipulate one's reality, not just a better pair of glasses. It allows users to engage with the physical world in new ways rather than rely on virtual reality for such possibilities. The existing Reality Editor already enables this behavior, as explored in depth in Heun's *Smarter Objects*. Rather than pursuing a question on this issue, I introduce this point to build up the question in Chapter 3.3, which I will explore in more depth.

3.3. Customize networks of smart objects with emergent behavior

The world is becoming "smarter" as it becomes more connected. There is a proliferation of sensors, actuators, and networked objects distributed throughout the physical spaces people live in, data from these objects could be used for many things. While the existing Reality Editor allows users to define connections between these Smarter Object, there is still a definitive lack of forming meaningful connections between objects in our lives. For example, a user might live in a house with a "smart" TV, alarm clock, lighting system, refrigerator, thermostat, security camera, and door lock, but these objects are relatively "dumb." Even when the existing Reality Editor makes them smarter by teaching them to talk to one another, they cannot be customize in how exactly they interact. They cannot help the user by responding to certain conditions with intelligent responses, or by collaborating to create routines of connected behavior. This thesis aims to solving this shortcoming by answering: How can we build functionality into the Reality Editor that allows users to build up intelligent networks of objects from unintelligent objects in an accessible and extensible way?
3.4. Collaborate to edit, share, and remix deployed creations

These deployed networks of logical interactions between objects also need to be created by someone, just as the AR interfaces need to be created. So just as we focus on the AR authoring pipeline, we must recognize the usability and accessibility of the network creation process as a hurdle to turning these systems from a collection of dumb objects into something really smart. Since we live in shared spaces, it comes to mind that many people might contribute little bits of intelligence to the system. In sum, you will get something really smart. In this thesis, I hope to explore, *How can we encourage and facilitate a collaborative editing environment where users can come together to augment their realities for the betterment of the group?*

3.5. Control one’s data privacy and provide adequate security

There is a final hurdle in making this a ubiquitous, useful system. People need to trust it. Even with good content available, and useful networks easy, possible, and accessible to construct, the system won’t accomplish anything if a lack of trust in prevents users from letting their personal object data flow throughout the network. This data is even more sensitive than data typically shared online, because it physically related to their lives. One’s object usage patterns could be used for a new level of targeted advertising. A separate fright is that a criminal with knowledge of when lights and appliances are off could use it to enter a house when no one is home. In a shared space, one might also not want everyone to be able to view all one’s data or disrupt the networks that they have spent time creating. In this thesis, I hope to explore, *How can we build tools that ensure user’s data privacy and security and build their trust in the system such that they actually use it?*

The features of the Reality Editor that I built to address each of these questions are described in Chapter 4 and 5. Chapter 4 discusses the Reality Editor as a tool for browsing AR content, which is deeply connected to the first of my research questions, as posed in Chapter 3.1. Chapter 5 goes into depth on the design and implementation of the Logic Crafting system and auxiliary tools, which answers the remaining three
questions from Chapters 3.3, 3.4, and 3.5. Chapter 6 evaluates the success of Chapters 4 and 5 by showcasing a variety of prototypes and use cases that were made possible by the technological improvements. Chapter 7 synthesizes this work to answer the above questions.
4. Augmented Reality Content Authoring and Browsing

As described in Chapter 3, the goal to extend the Reality Editor with "a comprehensive ecosystem for building, deploying, browsing, and customizing bidirectional AR content" relies in part on having a powerful but accessible method of generating AR content, placing it into space, and viewing it on demand.

At its core, our solution to drastically simplify the AR content pipeline was to rebuild the system using standard web technologies. AR interfaces for objects can now be created in the same way as any page on the web, and any existing web authoring tools or technologies can be used to create sophisticated interfaces without needing to learn any new specialized engineering techniques or processes (such as the proprietary C++ methods with which the application was previously written). The Reality Editor becomes a web browser for the physical world, rather than a closed system for interacting with a small set of engineered objects.

The decision to switch to web, rather than any other technology, was motivated by authoring, compatibility, and scalability issues. Web makes AR content easy to author and deploy, because it is supported by a robust set of authoring tools, many of which don’t require specialized engineering knowledge. Web makes AR content compatible with a range of existing technologies and services, because of the ability to communicate with a rich ecosystem of frameworks and APIs over standard internet protocols, such as HTTP. Finally, web makes AR content scalable, because it has been tested over decades to deliver content to millions or billions of clients.

Authoring, compatibility, and scalability are directly tied to the key metrics of connectivity that this thesis attempts to optimize. Accessible content authoring means more content that can be connected with. Compatibility means that a wider range of services can be connected with and made use of when building interfaces. Scalability means that the system will continue to work even when huge numbers of clients are connected to a
huge number of resources. Each of these points is important for maximizing the number of connected objects, clients, and AR interfaces.

4.1. AR Browsing (iOS Web View Bridge)

With these motivations in mind, we transformed the Reality Editor into a mobile AR web browser. Specifically, it is a native iOS app containing a web browser (a "Web View") that renders HTML content as an AR layer on top of the camera stream. On a high level, the app 1) uses native SDKs to download AR interfaces discovered in its local wifi network, and 2) uses the Vuforia SDK [34] for image recognition, which extracts the transformation matrices from recognized markers and 3) passes that information into the Web View for rendering the interfaces correctly in space. These interfaces are rendered as HTML iFrames with CSS3 3D Transforms applied to position them correctly on the marker. To do so, we need a bidirectional communication channel between the native application (written in C++ and Objective-C) and the interfaces rendered in the Web View (written in JavaScript). To enable this communication between the iOS app and the embedded web app, I wrote a set of classes that bridge between the two applications. We then moved as much code as possible into the embedded web app, because the only code that needs to be native are the parts that access iOS SDKs, such as the object downloader (which uses the openFrameworks ofxNetwork addon [35]) and the image recognition (which uses the Vuforia SDK).

The embedded web app is supported by part of the core iOS SDK. Native Objective-C applications support a class called the UIWebView which provides a window that can load local and remote web pages. It also provides some auxiliary methods to interact with the web content, such as the navigating the browser history or executing a script on the page. This provides communication in one direction: it lets us run JavaScript using Objective-C. The UIWebView also accepts an Objective-C delegate class that it notifies when certain events occur so that the app can respond appropriately with custom logic and actions. One such event is called shouldStartLoadWithRequest, which triggers when the web content sends a new request to the browser, such as to load a new page.
We want: JavaScript $\leftrightarrow$ C++

UIWebView enables: JavaScript $\leftrightarrow$ Objective-C

Objective-C++ enables: C++ $\leftrightarrow$ Objective-C

Resulting solution: JavaScript $\leftrightarrow$ Objective-C $\leftrightarrow$ C++

Table 2. An additional message-passing layer is required to communicate with the embedded web application.

We can use this to trigger events in the native app by making page requests from the JavaScript contents. This provides the other direction of communication: it lets us run Objective-C using JavaScript.

The implementation is complicated from the fact that the native Reality Editor application is written with a C++ framework called openFrameworks, rather than with Objective-C, so we cannot directly access this communication channel (see Table 2). The solution is my set of classes that provide a bridge between JavaScript and C++, using Objective-C++ [36] as an intermediary. Their fields, methods, and dependencies are highlighted in Figure 2. Integrating this bridge into a project is explained in Chapter 4.1.1. The details of the implementation of each class is explained in Chapter 4.1.2.

4.1.1. Integration

While the class definitions shown in Figure 2 may not appear to be a structurally simple communication channel, they are designed to have a very simple interface with the main application class. In this case, they integrate with the realityEditor class, which runs the main application processes of the Reality Editor iOS application, but they could be used to add an ofxWebViewInterface instance to any C++ class in an iOS application.

We make use of the Delegate design pattern found in Objective-C, “in which one object acts on behalf of, or in coordination with, another object” [37]. The delegate must conform to its object's protocol by implementing certain methods. With this contract in place, the object can send messages to its delegate to handle decisions about its state.
In the header file, realityEditor.h, proceed with steps 1 and 2:

1. Add ofxWebViewDelegateCpp as a superclass of the realityEditor class. This lets it receive and override the handleCustomRequest method:
   
   ```
   class realityEditor : public ofxVuforia_App, ofxWebViewDelegateCpp
   ```

2. Declare a variable for the Web View in the header file:
   
   ```
   ofxWebViewInterface interface;
   ```

In the implementation file, realityEditor.mm, proceed with steps 3 and 4:

3. Initialize the Web View interface in the constructor. This passes the realityEditor instance as the delegate of the Web View, linking its handleCustomRequest method to the Web View. We also load the index.html file into the Web View, which is stored locally and provides the entry point to the web application:
   
   ```
   interface.initializeWithCustomDelegate(this);
   ```
4. Listen for and handle requests from the Web View by implementing the delegate method:

```cpp
void realityEditor::handleCustomRequest(NSString *request, NSURL *url) {
    if ([request UTF8String] == "test1") {
        cout << 'test' << endl;
        // run the C++ code to respond to JavaScript "test1" command
    } else if ([request UTF8String] == "test2") {
        // ... handle each command separately ...
    }
}
```

The handleCustomRequest method in realityEditor.mm can now be triggered with a payload of "messageContents" by running the JavaScript command `window.location.href = "of://messageContents"`. This makes a page request that we intercept and handle with C++ rather than handling it in the browser. Conversely, we can inject JavaScript into the Web View with the C++ command `interface.runJavaScriptFromString("console.log('test')")`

4.1.2. Implementation

Such a simple interface with the embedded application required significant system design to implement, as hinted at by the classes in Figure 2. The implementation details of each class is explained below.

The C++ class running openFrameworks can’t be the direct delegate of the Web View. Instead, we need an intermediary Objective-C class that can be the delegate of the Web View, and which can also talk to the C++ openFrameworks classes. This is what ofxWebViewDelegateObjC is for. The sole purpose of this Objective-C++ class is to receive these Objective-C messages from the Web View and relay them to the ofxWebViewDelegateCpp class, which is written in C++. It provides this functionality by conforming to the UIWebViewDelegate protocol, specifically by implementing
shouldStartLoadWithRequest [38], where it then calls handleCustomRequest on its
ofxWebViewDelegateCpp delegate with the line:
[self delegate]->handleCustomRequest([request.URL host],
request.URL)

ofxWebViewDelegateCpp is an abstract class that the main realityEditor class can
inherit to implement the handleCustomRequest method. At this point, the functionality
will work for a UIWebView that sets a new ofxWebViewDelegateObjC instance as its
delegate, which in turn has a reference to the realityEditor class as its own delegate.
There are a few steps to set up such a UIWebView correctly, which I encapsulated in a
wrapper class called ofxWebViewInterface (named with the "ofx" prefix conventional to
classes compatible with openFramework). This class provides some convenience
methods for loading content, showing and hiding the Web View, and attaching the C++
delegate (some of which were omitted from Figure 2 for brevity). It also serves another
purpose: providing backwards compatibility for iOS 7, which uses UIWebViews for
embedded web content, while letting iOS 8+ systems make use of the similar but more
powerful WKWebView class, which has improved rendering speeds but a slightly
different API.

To allow either type of Web View, ofxWebViewInterface has an instance of a
UIWebView object (uiWebViewInstance), and a WKWebView object
(wkWebViewInstance) that it conditionally initializes and subsequently uses depending
on the OS version. Most logic in the ofxWebViewInterface class occurs in its
initializeWithCustomDelegate constructor, which initializes the correct Web View object
and sets up its connections between the ofxWebViewDelegateObjC and
ofxWebViewDelegateCpp. It has additional methods that wrap the standard functionality
of a Web View, such as loadURL, which have additional logic to adapt to the differing
APIs for UIWebView and WKWebView.

This completes the necessary architecture changes to support a web-based mobile AR
browser. At this point, the communication channel with the web app lets us move almost
the entire application code into the Web View, enabling us to then render each AR interface using standard HTML, CSS, and JavaScript, rather than relying on C++ content specifically engineered to work within our system. As previously stated, this is important for authoring, compatibility, and scalability reasons, which are prerequisite to making AR browsing a ubiquitous part of life. Empirically we have found this switch to web to be a good choice, as we have been able to quickly prototype web content and build on top of existing tools that we would have otherwise been bogged down trying to reimplement in our C++ framework. Example prototypes are described in Chapter 6.1.

4.2. AR Authoring and Deployment

In addition to rendering existing web content, we have a workflow to allow developers to author custom HTML interfaces with AR capabilities, and we did not even need to build our own authoring tools to do so. Interfaces can be built with any content creation software and processes familiar to web developers. This could be as simple as writing vanilla HTML, CSS, and JS files. Developers can also use their favorite web frameworks such as React, jQuery, Bootstrap, and Meteor. Creating AR interfaces is even accessible to those without coding experience, who can use visual tools like Adobe CC to prototype and deploy working webpages. Even mockup sketches can be previewed as images displayed in the AR browser.

While these examples stress accessibility as the motivation for web-based AR authoring, but another key motivation is how powerful the web is. Interfaces can access web APIs as easily as any website, which lets them tap into a huge number of available datasets and remote services. Current prevalent competitors for AR authoring include the game engine Unity [39], which many developers rely on to build graphically-rich AR experiences. However, we argue that building with web technologies is superior for a number of reasons.

1. Web is standard, distributed, and doesn’t rely on the financial success of any one company or institution (e.g. Unity or Apple).
2. Web has proven to be a reliable technology across time, as the first webpage (built over 25 years ago) can still be visited and rendered today in a modern browser [40].

3. Web has a huge community of knowledgeable developers who would not need to be retrained to build AR content, and an established set of tools which can be repurposed for AR. This is a financial incentive for companies deciding which platform to develop with.

4. Web has a variety of included features and protocols that developers do not need to invent novel solutions for, such as security features provided by HTTPS.

5. Web continues to grow as a platform, and supports ever more sophisticated content. Rich 3D content that once required a dedicated game engine like Unity can now be rendered in browser using WebGL [41] and other emerging technologies, which leaves little added benefit to using a separate platform.

Convinced by the promise of web, our team employed a number of processes to produce example interfaces that demonstrate the web's capabilities for AR browsing. Empirically, we have found this change in technology to be a good choice; we have been able to quickly prototype, view, and interact with a diverse range of AR experiences. These range from an animated 3D turtle, to a dynamic table of grocery nutrition comparisons, to an x-ray overlay for an art gallery, which can all be seen in Chapter 6.1.

4.2.1. Hybrid Object API

My collaborators created an API that these interfaces can utilize for more complex behavior. It allows interfaces to communicate with the rest of the system with subscription and broadcasting of events. These include: proximity (knowing how close the user is to the marker), visibility (which interfaces are currently within the viewport of the user), fullscreen mode (view content flat over the entire screen rather than in an AR container), global broadcasting of arbitrary data between visible interfaces, and matrix data (getting updates about the complete transformation matrix between the marker and
the user, for custom graphics behavior). Overall, this API lets developers create complex interactions between multiple interfaces that enables systems of responsive, interacting AR interfaces without specialized programming knowledge. I did not make direct contributions towards this API, but it is utilized heavily in some prototypes in Chapter 6.

4.2.2. Unconstrained Interface Positioning

The final step in deploying an HTML interface onto a marker is to define its spatial properties such as its size and position relative to the marker. In other systems, this typically requires knowledge of 3D graphics, either in the form of working with a 3D scene editor like Unity, or programming the transformation matrices directly with code. But developers are typically much more familiar with working in 2D space. We built direct manipulation tools into the Reality Editor so that this 3D positioning can be done without coding, with nothing more than one’s spatial intuition. A collaborator previously built an AR Repositioning mode within the native app. While this mode is toggled on, dragging an interface will translate it within the 2D plane of its marker, and pinching will scale it up or down. We moved this functionality into the Web View, but it is still fairly limited in how you can position the interfaces. For example, you cannot pull them out along the z-axis so they float in front of the marker. And you cannot rotate them along multiple axes so that they appear at a tilt or completely orthogonal to the marker.

To solve this without requiring any knowledge of 3D graphics, we implemented a direct manipulation tool called Unconstrained Repositioning. In this mode, a user adjusts the translation and rotation of their interface by physically moving their phone in space to a parallel orientation. To initiate a reposition, the user taps and holds on the interface. This locks the relative transformation between the phone camera and the interface. Moving the camera in space will then push, pull, translate, and rotate the AR interface in 3D space such that it maintains this original relative transformation to the camera. Releasing the tap will set the interface in this position, and the phone can be moved
again without changing its position. In this manner, any user who knows how to build 2D interfaces can place their creations in a physical space using only their spatial intuition. My collaborator implemented the core functionality of this mode, which uses the projection matrix of the camera to affect the transformation matrix of the selected interface. My contribution, however, belongs to the domain of computational geometry. It tackles a consequent, and perhaps more interesting problem, which is to provide visual feedback to the user about their repositioning. Specifically, to provide feedback to help users see where the z=0 plane of the marker is in relation to their repositioned interface. This is important because there are unintuitive visual effects if the interface gets placed “beneath” the marker (if its corners all have z < 0). In this case, the interface still gets rendered on top such that it obscures the marker, but its perspective behaves as if it is beneath it. To help users recognize and prevent such a situation, I created a visualization that gives the interface a colored overlay while you are repositioning it. The portion of the interface in front of the marker plane is colored blue, while the portion behind the marker is colored purple, as seen in Figure 3. This provides realtime visual feedback to help users position their AR content.
I implemented this in a function called drawPlaneIntersection, which calculates the points on the interface overlay that intersect the marker plane (z=0), and divides the overlay into two regions that are colored accordingly. The pseudocode for this algorithm is outlined in Figure 4.

The algorithm begins by assuming everything is in front of the z=0 plane, and colors the entire rectangle blue. It then tries to find two pairs of adjacent corners that fall on opposite sides of the z=0 plane. If there are not exactly two of these, then the interface doesn’t cross the marker plane and so everything will be the same color. Otherwise it solves a linear equation for each of the two oppositeCornerPairs to find the point between them with z=0. These two points are the endpoints for the line segment that divides the purple and blue regions of the overlay. The endpoints and the corners now provide polygons within which the colors can be rendered into an HTML5 canvas element. In the case where the interface is entirely behind the z=0 plane, there is no line segment to divide the regions, but because the corners all have z < 0 the overlay will be colored entirely purple.

```
drawPlaneIntersection
{
  - draw blue over everything
  
  - in clockwise order, get all corners (x,y,z)
  - look at every pair of adjacent corners
    - if they have opposite z values
      - add to oppositeCornerPairs
    
  - if number of oppositeCornerPairs > 0
    - for each corner in oppositeCornerPairs
      - calculate linear equation to find x,y point where z=0
        - add point to interceptPoints
    
  - also add corners with z < 0 to intercept points (there will be 1 or 2 of these if any intercepts were found, or 4 if completely behind z=0 plane)
  - sorts interceptPoints clockwise
  - creates a stroke from sortedPoints
  - uses stroke to create a purple mask for that region
}
```

Figure 4. Pseudocode for unconstrained repositioning drawPlaneIntersection algorithm
4.3. Discussion

Improving the AR content authoring process involved a time investment to rewrite the system using the web as its core technology. This involved some engineering within the native app to establish a JavaScript to C++ bridge. The result is that content can now be built with a more accessible, compatible, and scalable technology.

Concrete examples of the success of these efforts can be seen in Chapter 6. On a conceptual level, the improved AR content authoring process improves two key metrics of the Reality Editor: usability and connectivity. The web-based interface production directly improves the usability for developers, as they can use their preferred techniques rather than being forced to adopt one provided by the system. It also improves the connectivity. By making interface creation more accessible, system adoption by developers will increase, and therefore increase the number of objects available for users to connect. This proliferation of objects is central to the success of the Reality Editor and its goal to make user's lives better by breaking down the boundary between the virtual and the physical.
5. Logic Crafting

Whereas the AR authoring improvements contributed to the usability and connectivity, they don't directly affect the capability of the system in terms of the potential behavior of Smarter Objects. The capability of objects connected by the system is currently limited in the simple way that they interact. Data is transmitted directly from one object to another with no processing or logic in between. This is useful for triggering events in other objects, or using the value of a sensor to directly set the value of an actuator, but further application are impossible to create within the UI of the Reality Editor. To accomplish more complex data processing than sending the value of one IO node to another, developers currently need to reprogram the server. Custom behavior can be engineered into an object by modifying the server, or in some cases by making extensive use of the Hybrid Object API and including logical computations within the visual interface code (thereby losing a clean separation of the model and view). Neither method allows an end user of the system to customize data processing with additional logic in realtime. This type of functionality is necessary for the average user to build truly intelligent networks out of general-purpose objects, which can be configured to meet their exact needs.

To solve this problem, we introduced simple logical operators that can be visually added to the node network via direct manipulation interactions. These logical operators can be connected in a flow-based programming environment we call Logic Crafting. Each functional collection of operators is deployed into the system within a module we called a logic node. Logic nodes have a similar architecture to IO nodes, but are purely virtual rather than being associated with a hardware element, and can be configured to manipulate the data that flows through them.

The design and implementation of Logic Crafting was an extensive process that required the majority of time spent on this thesis. First was the initial brainstorming, design, and paper prototyping of the user flow. Next was the front-end implementation, which involved solving problems such as rendering, algorithms, usability, data
structures, and event handling. With the front-end in place, it needed to be integrated with the back-end. Finally, I built web service and data security features on top of the Logic Crafting platform. An assortment of prototypes exploring the possibilities of Logic Crafting are explained in Chapter 6.2.

5.1. Design and User Flow

The process of adding these logical operators involves interactions in the Node View and in the Crafting View. In the Node View, users create logic nodes and link them to surrounding IO nodes (see section 4.2.1). Tapping on a logic node enters the Crafting View, which allows it to be configured with a set of linked logic blocks (see section 4.2.2).

5.1.1. Node Interactions

Let there be two existing IO nodes called A and B that a user wishes to connect with an intermediate program. This program could be anything, but for the sake of a concrete example, say that B should activate 5 seconds after A is at least halfway on. The user flow for creating such an interaction involves actions in the Node View and the Crafting View.

The user navigates to the Node View using the sidebar buttons shown in Figure 5. The first button displays the Interface View and the second the Node View; the third is a pocket where logic nodes can be dragged out out; the fourth shows the settings menu and the last freezes the AR video stream. In the Node View, the user adds an empty logic node, L, by dragging one out of the pocket button and placing it in on an object visible in the AR

![Figure 5. Adding a link from IO node "A" to logic node "L" displays the port selection panel, where the user chooses a color for this link to identify it within the Crafting View.](image)
viewport. The draws a link from A to L, and then from L to B. When the user hovers over L while creating the link to it, they are prompted to select one of four colors for the link: blue, green, yellow, or red, as seen in Figure 5. The selected color is used within the Crafting View to distinguish multiple input signals so that they can be processed and combined accordingly. In this case there will only be one input to L so any color will suffice. Similarly, the user chooses a color for the link leaving L, which allows a logic node to output multiple distinguishable signals. Data will now be sent from A to B, but will be processed by the logic node in between.

5.1.2. Logic Crafting Interactions

By default, L has no program so it will just output the same values to B that it receives from A. To change this, the user taps on L to open its Crafting View. The Crafting View contains the resources and interactions necessary for users to create small programs that affect the flow of data from one node to another. This primarily consists of a 4-by-4 grid where resources can be placed, which we call the Crafting Board. An empty crafting board can be seen in Figure 6. The resources users place into the board are called logic blocks, which occupy cells in the grid and can be connected with links in a similar way that nodes can be connected in the Node View. There are a variety of logic blocks which each perform a specific operation on the data they receive. For example, a Delay Block occupies one cell and outputs its single input N seconds later. An Add Block, on the other hand, occupies two adjacent cells in a row, which each have an input that get added together and written to a single output. The Crafting View also contains a new sidebar with buttons specific to Logic Crafting.

Figure 6. An empty crafting board (left) and the block menu (right).

37
The top row of the Crafting Board consists of four input cells for the logic node, and the bottom row consists of its four output cells. Each column is color coded to match one of the port colors (blue, green, yellow, or red). Data that enters a colored port of L in the Node View gets routed to the corresponding crafting input cell. For example, if the user linked A to L via its blue port, incoming data will arrive at the blue input cell and propagate to a logic block placed in that location.

To construct a program out of logic blocks, the blocks must be selected from the Block Menu, which can be opened by tapping a button in the sidebar. The Block Menu has five tabs that can be selected to view blocks belonging to certain functional categories: blocks, events, signals, math, and web. Each tab displays a list of blocks (for example, those shown in Figure 6). Pressing and holding one of these blocks will pick it up, hide the Block Menu, and allow the user to place the block in any open cell of the Crafting Board. To build the example program, the user opens the menu, locates the Threshold Block, and places one into the crafting board, then repeats with a Delay Block.

Once placed, the blocks can be linked together by drawing lines between them. Blocks can also be linked to the logic node’s input and output cells in the same way, even if they aren’t occupied by a block. Assuming the user chose blue and red for the input and output ports, respectively, the user now draws a link from the blue input cell to the Threshold Block, then from the Threshold Block to the Delay Block, and finally from the Delay Block to the red output cell. This strings together the functionality of the blocks and connects to the ports from the Node View. The last step for this program is to customize the settings of individual blocks, for example setting the Delay Block to use a time interval of five seconds. The user taps on the Delay Block to open its Block-specific settings menu, and interacts with a slider element to set the time interval.

When the programming is complete, the user taps the back button in the sidebar to close the Crafting View and return to the Node View. At this point, the user has successfully added custom logic to the network that will activate any time data is
emitted from A. The same process can be used with more IO nodes and a wider variety of logic blocks to build increasingly complex functionality.

5.2. Features and Implementation

While my collaborators handled the majority of the Node View implementation, the bulk of my thesis consists of the implementation of the described user flow within the Crafting View. I constructed the entirety of the crafting GUI with the exception of the settings menu. My contributions to the back-end were less comprehensive, but involved significant design decisions about syncing the state of the logic node with the server. I present the front-end and back-end engineering details of a novel grid-based data flow programming environment for AR and IoT interactions.

5.2.1. Front-End Implementation

The most significant software development of this thesis consisted of implementing the Logic Crafting GUI. Based on our conversations and paper prototypes, my collaborator created a six page visual specification with dozens of figures describing the intended user experience for this view. I converted these static mockups from Illustrator into a dynamic user interface built with HTML, CSS, and JavaScript, as seen in Figure 7. This process involved many design decisions that could not be predicted by static mockups. These include animating the links, moving blocks, and how the design works given the

Figure 7. Logic Crafting visual specification (left) compared with the resulting implementation (right) Note that this isn't the final implementation because there was another design iteration.
transition between the Node View and Crafting View. Partially as a result of these decisions, a second design iteration is presented in Chapter 5.2.1.7.

I prototyped the core interaction of the crafting board outside of the Reality Editor in a browser environment on my laptop to speed up the development process. The tradeoff was needing to later refine the interactions for a touch interface, and to be consistent with the context of the Reality Editor. This mostly involved implementing a 4-by-4 grid of cells with margins between them, and blocks that can be added to cells and connected with links. I broke down the process into a number of subproblems to solve, including:

1. Designing data structures to represent the Crafting Board model (see 5.2.1.1)
2. Structuring the DOM correctly to render the grid, blocks, and links (see 5.2.1.2)
3. Designing a routing algorithm to find legible paths through the grid (see 5.2.1.3)
4. Managing user interactions for moving, connecting, and cutting (see 5.2.1.4)
5. Designing an interactive block menu that pulls data from the server (see 5.2.1.5)
6. Making the visuals more understandable and consistent (see 5.2.1.6)

5.2.1.1. Data Structures

Before any subproblem can be solved, the system needs a sufficient data model to store the necessary information in a convenient way to be queried for algorithmic and rendering purposes. As described in the user flow, each logic node has a crafting board and a set of blocks and links. Accordingly, we define the Block class, the Link class, and the Grid class (which supports a number of grid-based operations necessary for the crafting board). These classes and their dependencies are diagrammed in Figure 8. I will explain the purpose, information structure, dependencies, and helper methods of each class. Defining these data structures allows the crafting board to be programmed at a higher level of abstraction. Rather than working with raw pixel coordinates in a rendering context, its functionality is neatly wrapped into the grid, blocks, and links.
CellLocation

A CellLocation instance simply contains a (column, row) coordinate pair. It is useful to encapsulate these in a class to associate the coordinates at a conceptual level.

Cell

A Cell instance has a CellLocation and a wide assortment of helper functions. There are two types of cells: block cells and margin cells. Block cells exist at a grid location where a block can be placed, as indicated with color in Figure 7. Margin cells are the smaller gaps between them, which cannot hold blocks but are useful when finding paths for links. A cell's helper functions include determining whether it is a block or margin cell, counting how many horizontal and vertical link routes pass through it, and looking up the block placed at its location (if one exists).

Figure 8. Logic Crafting data structures.
Grid

There is one Grid instance per logic node, and it contains the highest-level information and methods. It contains a complete list of all the cells that compose it, as well as the pixel width and height information necessary to render that set of cells onto the screen. It has methods for calculating relationships between other objects, especially for calculating the link routes, and for rendering everything to the screen, which requires converting from rows and columns to pixel coordinates.

Route

A Route instance has a list of cells that compose a path from a start to a destination in the grid. It has another list with a subset of those cells that contain corner points of the path, which is useful for rendering, and it has an object called pointData that gets populated with pixel coordinates representing the actual path to draw on the screen.

Block

A Block instance has information about its position and size within the grid. Its size specifies the number of columns it occupies, which are referenced by their item indices. A logic node has a list of blocks that it can render using this information.

Link

A Link instance has references to the blocks it connects and the item indices of each connected block. It also has a route instance that gets recalculated whenever something in the grid gets moved, added, or deleted. A logic node has a list of links that it can render using their route instances.
5.2.1.2. DOM Structure

Following a model-view-controller design pattern, we need to separately design the data model and how to render the view, and then provide controller logic for how they influence one another. The data structures set up the model, so the next step is to decide how to display the view.

I considered several methods for rendering the Crafting Board, which includes the background grid, the blocks, and the links. The first was to render everything inside a single HTML `<canvas>` element, as the Crafting Board is graphically rich and seems to require the custom drawing capabilities of a canvas. Particularly, the links seem to need a canvas to animate dotted lines in piecewise paths. But by default, a canvas just renders pixels to a graphical context with no associated object, which is too primitive to easily connect with our data model. To get around this, my first prototype used the Easel.js framework [42], which adds “a full, hierarchical display list, a core interaction model, and helper classes to make working with Canvas much easier.” For example, rendered shapes can be associated with container objects which can be moved around, grouped with other containers, and attached to events to build modular interfaces inside a canvas element.

While this DOM structure worked, I realized I was making the solution overly complicated because most of the elements can be displayed and interacted with more easily outside of a canvas context. The blocks, for example, are just colored rectangles that can be rendered as a styled `<div>` element and subsequently moved around without needing to clear and redraw the entire canvas. I was able to display everything using divs and CSS, except for the links, which I still render in a canvas context (but without the need for Easel.js) arranged between the grid background and the blocks.

5.2.1.3. Routing

With the model and view in place, some logic is needed to tell the view how to render a particular set of data structures. This is fairly simple for the block instances; for each
block in the logic node, add a child div styled with CSS to the crafting board, set its location based on its corresponding cell, and customize its contents with its name or icon image.

It is less clear how a logic node's set of links should be displayed in the view. Ideally, each link would find the best path from its start block to its end block through a potential maze of other blocks and links, so that it avoids other blocks and it distances itself from other links so that each can be clearly seen. I implemented a routing algorithm that finds an acceptable set of points in the canvas to draw such a path. The routing algorithm can be broken into two subproblems: find a valid path, and then making it legible. The valid path problem requires finding a set of block-free cells in the grid that connect the exit of the first block with the entrance of the second. The legible path problem requires determining a set of coordinate points within those cells to draw the line segments such that overlaps are minimized. For example, if we drew every link in the center of each cell, then all vertical lines going through one column would be drawn collinearly and it would be impossible to distinguish where one link started and another ended. This would be a valid path but not a legible one.

I considered a number of techniques for each of the subproblems. A generalized pathfinding algorithm such as A* would solve the first problem nicely, given a concise

```plaintext
calculateLinkRoute
{
    - start block cell
    - if link goes down
      - go down as many cells as possible before hitting a block
      - go horizontally to margin cell closest to end column
      - go down to margin row above end block
    - else (link goes up or same row)
      - go down one cell
      - go horizontally one cell towards end column
      - go up to cell in margin row above end row
    - go over to cell directly above end block
    - end block cell
    // note: several edge cases excluded for readability
}
```

Figure 9. Pseudocode for the Valid Path Problem
representation of the grid as an adjacency map. Even better solutions could build on the ideas of electronic design automation, which use a variety of routing algorithms to design integrated circuits and printed circuit boards [43]. However, I realized I could exploit some constraints of the grid to yield an even simpler and more efficient algorithm that doesn’t need to search the entire grid. It relies on the fact that there are always open cells in the margins between each block row and column, so the most complex path will have at most have four corners: two to turn from the start block towards and down the margin, and two to turn out of the margin and into the destination block.

Accordingly, the valid path algorithm calculateLinkRoute (which can be seen in Figure 9) starts with the cell beneath the start block, and then goes over to the closest margin cell in the same row (on the side closer to the destination block). Then it goes up or down to the row directly above the destination block, and finally goes horizontally over to the block directly above the destination block. There are a few edge cases excluded from the pseudocode. For instance, if there aren’t any blocks placed between the start and end blocks in their same column, the algorithm skips the steps in the middle and just goes straight to the destination. There are also edge cases if either block is in the top or bottom row, but it does not get much more complicated and the above algorithm handles the majority of cases.

The calculateLinkRoute algorithm solves the Valid Path Problem and yields an ordered list of cells that the link spans. It can be drawn by connecting line segments between the centers of each cell in the path, as seen in Figure 10. In this example, there are nine
determineMaxOverlaps
{
  // decreases future overlaps of links in the grid by sorting them left/right
  - for each column c
    - sorts list of links by left-to-right horizontal travel distance
    - filters out links that don’t include any cells in column c
    - for each cell in column c
      - count number of links vertically passing through cell (includes corners)
      - set maxOverlappingVertical to max of these values for c
  // vertically sorts them so that links starting near horizontal center are below
  // those starting near edges, so they don’t overlap
  - for each row r
    - sorts list of links vertically based on order in horizontallySortedLinks
      such that those in middle horizontally are lower, those near edges are higher
    - for each cell in row r
      - count number of links horizontally passing through cell (includes corners)
      - set maxOverlappingHorizontal to max of these values for r
  - return list of sorted overlap information for each row and column
}

Figure 11. Pseudocode for first step of the Legible Path Problem.

links in the crafting board, which causes many cells to have multiple links in them. Not only is it important for legibility that these links be visually spaced out, but it is also important for our future link deletion mechanism, which is modeled after link deletion in the Node View. To delete a link, a user “cuts” it by drawing a line across it. If all line segments are drawn on top of one another, it would be impossible to delete one link without deleting its neighbors as well.

For these reasons, I designed a two-part legibility algorithm to solve the Legible Path Problem. The legibility algorithm starts with the cells computed for all links from running the calculateLinkRoute algorithm on each, and performs two additional routines on this data, as seen in Figure 11 and 13.

Figure 12. Visualization of output from determineMaxOverlaps. V and H are the number of vertical and horizontal segments in a cell.
The first is determineMaxOverlaps, the pseudocode for which can be found in Figure 11. This does two things. It counts how many routes overlap vertically and horizontally in each cell, and the maximum of each of these values for each row and column, which will later be used to compute the best spacing between links. It also sorts the list of links according to their destinations, to minimize overlaps. For example, links going to the left are ordered before those going to the right, so that they can be added left-to-right and not cross. A visualization of the route overlap counts can be seen in Figure 12.

The second routine in the legibility algorithm takes this sorted metadata about the grid and uses it to stagger the link positions within their cells. This is accomplished with a function called calculateOffsets, described in Figure 13. It draws each link to the grid in the sorted order, keeping track of the number of links drawn so far in each row and column, and how that compares to their maximum overlapping number of links. The ratio of previously drawn vertical segments compared to the total number expected in a cell is used to place the link horizontally. For example, if there are 8 vertical links going through a cell, and this is the sixth one to draw, it will be placed 75% between the edges of the cell. A similar calculation is used for the link’s vertical position, but using the number of horizontal segments in the row rather than vertical segments in the column. This spaces links evenly apart, and the sorting from the previous routine was carefully

```
calculateOffsets
{
  - for each column c
    - get the route overlap information for c, generated by determineMaxOverlaps
    - calculate horizontal offset between links in c, given as columnWidth/(maxOverlap+1)
      - for each link in that column, in the sorted order from determineMaxOverlaps
        - set its x position within c = horizontalOffset * numRoutesProcessed

  // same for rows and vertical offsets
  - for each row r
    - get the route overlap information for r
    - calculate the vertical offset between links in r
      - for each link in that row, in the sorted order from determineMaxOverlaps
        - set its y position within r = verticalOffset * numRoutesProcessed
}
```

Figure 13. Pseudocode for the last step of the Legible Path Problem.
selected to minimize the number of links that cross each other in the same cell. The result is a system that clearly displays how many links are in each cell, and where the user can follow the path of a link from start to end. The example from Figures 10 and 11 can be processed with this algorithm to produce Figure 14. This figure actually omits the full sorting information from calculateMaxOverlaps. To see the effects of this sorting, compare the overlapping routes in Figure 14 with the cleanly organized routes in Figure 15.

Figure 14. Resulting routes from completed Legible Path Problem, with the exception of horizontal sorting to minimize overlaps.

Figure 15. Final link rendering algorithm results. Dots are animated to move from the start block to the end block, showing the directionality of each link.
With validity and legibility complete, we can still improve the links’ visual style. Not only can this make them more visually pleasing, but it can make them more distinguishable, it can encode information about the link destination, and it can make the transition between the Crafting View and the Node View more seamless. A link in the Node View is represented with an animated, dotted line, which fades to the color of the logic node’s port that it is connected to. To improve the internal consistency of the user interface, the same styling is applied to the links here, which now fade from the start column’s color to the end column’s color. This is implemented by representing the color in HSL format, and interpolating the hue between the two columns. Rather than drawing a single line on the canvas, it now draws a large number of 10-pixel-diameter circles along the path, each with a separately interpolated color. They are animated each frame by clearing the canvas and redrawing the circles with an incremented offset that eventually resets. This animation makes it easy for users to tell which is the start block and which is the destination of each link, as the dots flow to the destination. Logic Crafting is a data flow programming environment, and the flowing dots also enhance the visual metaphor of flowing data.

5.2.1.4. Touch Interaction

As referenced in the user flow chapter, the user interactions inside the Crafting View are modeled after the existing user interactions in the Node View. The goal was to implement a consistent set of gestures that have predictable effects in both views. This consistency is a key usability feature because users won’t need to learn a new interaction pattern for the Logic Crafting, and won’t confuse users when switching between contexts. The main difference between the two is that the Crafting Board constrains blocks to a grid.

The number and variety of these interactions would result in a mess of overlapping event listeners unless carefully architected. To support these interactions I created a finite automata representing the current state of the user interaction, which is diagrammed in Figure 16. Depending on this state and a new touch event, the finite
Figure 16. Finite state machine representing the current interaction state and touch events that transition between them.

Automata will transition to a new state and the user interface will update accordingly. The actions that cause transitions are pointerdown, pointermove, and pointerup (or pointercancel, which triggers the same event), as well as a timeout event that triggers after a certain delay.

The automata begins in the NONE state. A “down” event in this state will transition to TAP if it was on a block, or CUT otherwise. While in CUT, a line will be drawn from the tap’s start coordinate that lets the user cut links, and an “up” event will delete the selected links and return to NONE. While in TAP, if the user releases quickly it will open the settings page and return to NONE. If they “move” quickly it will transition to CONNECT and begin drawing a link, which gets created and returns to NONE on an “up” event. Otherwise in TAP, if sufficient time passes without moving, a timeout will occur and the automata will transition to HOLD, which picks up a block and visualizes this to the user, but doesn’t actually disrupt the block from its position in the crafting.
board. An “up” event will place it back down and return to NONE, but a “move” event will transition to MOVE, which removes the block from its cell, drags it around, visualizes new paths for any connected links, and snaps it into the grid if it is sufficiently close to an open cell. An “up” event will return to NONE and attempt to place the block in the closest cell, otherwise it returns to the cell it started from.

This completely describes the user interaction model of the Crafting View, and the implementation is a straightforward transcription of these states and transitions. The only complication that arose was determining where to attach the touch event handlers to minimize complexity. The background canvas, the input and output cells, and the blocks all potentially need their own event handlers to be added and removed, and to decide whether to capture the events or let them propagate to objects beneath them. I found it simpler to only attach event handlers to a single transparent div on top of the crafting board that that covers the entire screen, called the logicCraftingEventDiv. I then include centralized logic that queries the grid using the touch position to determine what objects exist under that position, and processes the event with that object appropriately. A final note is that we must use a polyfill to access the Pointer Events Web API that lets the Safari browser implement touch events on mobile devices [44].

5.2.1.5. Block Menu

The Logic Crafting implementation described so far assumes that blocks have already been placed within the crafting board. I also fully implemented the Block Menu GUI, which loads a selection of blocks from the server and lets users drag them into the crafting board.

When the user taps on the Block Menu sidebar button, a div is added over the whole screen that contains the menu, which is a set of category buttons and a large block selection panel. It immediately makes a request to the server for the entire list of blocks, which are returned as json data including the name of the block, the category it belongs to, and a url where its icon image can be loaded. It immediately attempts to fill the block
selection panel with blocks belonging to the first category. To do this, it filters out blocks from the blockData that don’t match the current category. It then queries the server for each of their icon images in turn, and renders each block as a div styled with the same CSS as in the crafting board. It also displays the name of the block adjacent to it to disambiguate the icon. Tapping on another category button removes all children from the block selection panel and reloads it with a new set of blocks.

The Block Menu has a separate event listener from the crafting board, which listens for down, up, and move touch events on the blocks. When a block is dragged, the Block Menu is removed from the DOM, and a new Block instance is added to the logic node’s list of blocks with the selected block’s attributes. This block is rendered at the same position as the menu block to provide continuous visuals, and the user interaction state transitions to MOVE. This new block behaves exactly the same as existing blocks, except it doesn’t have a cell to return to so it gets deleted if dropped in an invalid location.

![Figure 17. Original Logic Crafting style (left), vs implemented restyle (right)](image)
5.2.1.6. Design Enhancements

The block menu implementation completes the crafting board functionality. After testing this with users, we iterated on the design to improve the user experience. The resulting changes are compared with the original interface in Figure 17. The main goal of this revised style is to blur the boundary between the Node View and the Crafting View. Keeping the user interactions consistent was the first step to bridging the divide between the two views and making it feel like a single cohesive application. But we encountered a problem where users felt disconnected from the AR application while in the Crafting View, which consequently made it more difficult to remember which IO nodes were connected with colored ports. To minimize the cognitive load of entering or exiting the Crafting View, we proposed a number of visual and functional changes that I iteratively implemented.

From a purely visual standpoint, my team created a new style guide for the crafting board that preserves the look and feel of the node level application, which I

![Image of block repositioning in the restyled GUI. As the block moves from the first column (upper left) to the second column (lower right), visual feedback is provided about the shifting links and whether the block is in an invalid (upper right) or valid (lower left) cell.](image)
implemented mostly with CSS adjustments. The overall style uses neon colored borders that contrast with the camera stream, and transparent backgrounds when possible to let the camera stream show through. The neon colors also go well with the links, which already have this color scheme. More prominent visual feedback is displayed for interactions such as moving blocks, as illustrated in Figure 18. A big visual change to connect with the Node View is changing the background to a semitransparent blur filter, which allows some of the AR camera stream to show, rather than blocking it all out with an opaque grey background. To do so, it applies a backdrop-filter: blur(30px) using CSS, which blurs strongly enough that it doesn’t distract too much from the foreground, but still gives a sense of connection between the crafting board and the rest of the application.

A useful extension of this semitransparent background is the Node Preview feature, which fades out the background entirely (along with most of the crafting board contents) when the user hovers over an input or output cell for a lengthy duration, as seen in Figure 19. In its place, they see the Node View, frozen in perspective the moment the user opened the crafting board, so they can see what is connected to the open logic node. This lets them quickly see and remember which IO nodes are connected to which colored ports. When they move their finger away from the cell, the entire crafting board

![Figure 19. To preserve the context between the Node View (upper left) and the Crafting View (lower left), hovering over the output cell causes the crafting board to fade away (right). This helps users remember which nodes are connected to which ports.](image)
returns. The hiding and showing of the crafting board is implemented as a CSS animation that smoothly fades between views rather than an abrupt cut. The whole point of this is to break down the abrupt boundaries between the two views, so adding the animations is important for its cognitive effect in addition to the aesthetic appeal.

5.2.2. Back-End Implementation

Much of the server-side implementation for the Logic Crafting was implemented by my collaborators. However, I was involved in connecting a number of my front-end components to their back-end representations. Specifically, I worked to upload and synchronize the state of the logic node as changes are made to its crafting board, and broadcasting those changes to other users to support realtime collaboration. I also made some design decisions about how to interface the logic node state with the IO nodes on the server. My team created the API and basic structure for logic blocks on the server, and I took this template for a default block and implemented about a dozen types of blocks. Of particular note are the web service blocks that I conceived, implemented, and built demos to showcase. Finally, I realized the lack of any data security or privacy features, and prototyped a locking mechanism for users to control access to their objects.

5.2.2.1. Realtime State Synchronization

Chapter 5.2.1 demonstrates the multitude of ways in which the front-end logic node state can be changed and visualized. This changing state within the Reality Editor application needs to be synced with the server so that the created programs are actually uploaded to the network of objects and affect the flow of data.

There are two apparent methods for sending this state to the server: snapshot-based synchronization or differential synchronization [45]. The snapshot approach would resend the entire state of the logic node every time anything in it is changed. This guarantees that the server will eventually have the same state as the client, as it will
only ever receive a state that matches a client’s exactly. The difference-based approach, on the other hand, would incrementally send only the changes in state (the "deltas") to the server, for example “link A was created” or “block B was moved to position C”. This reduces the amount of data that needs to be sent, as most of the state in the first approach is redundant.

I began implementing both methods, but eventually chose the snapshot approach. In practice, the state of a logic node is relatively small. At most, there could be 16 blocks and $16^2$ links. A block takes about 1500 bytes to encode as JSON data, and a link takes about 300 bytes, so the theoretical maximum size of each snapshot is bounded by almost exactly 100 KB. Even this would be acceptable to synchronize each time, but this overestimates the typical logic node by at least a factor of 10. The example from Chapter 5.1.2, for example, only has two blocks and three links, which could be encoded in roughly 4 KB, so the efficiency benefit of the difference-based approach is minimal. The simplicity and robustness of the snapshot approach outweighs the few KB per action that might be saved on average with the other approach. And unless we implemented a sophisticated transactional system to ensure the validity of the difference-based approach, it also is more prone to consistency errors. If one incremental change fails to send, the client will end up with a different state than the server and there will not be a clear way to recognize or resolve this. With a snapshot approach, however, even if one state fails to send, the server state will eventually be overwritten by an accurate client snapshot.

In short, I evaluated the demands of the system and made a design decision that prioritizes simplicity and consistency over efficiency. A future researcher may wish to implement a more robust transactional system that would allow for the efficiency gains of a difference-based approach without risking the correctness of the data.

Implementing this snapshot synchronization involved creating a new route on the server that accepts a post request with a logic node state serialized as JSON in the payload. Note that there is additional state cached in the logic node pertaining to the state of the
user interface (such as the grid, which elements are currently being interacted with, and a cache of the block DOM elements) that we strip away before serializing it. This data can all be repopulated from the logic node’s blocks and links when needed, and removing it lets us meet the efficiency estimates previously described.

In addition to uploading programs to the network, uploading the state allows users with separate devices to view the changing logic node. A consequence of the increased power of the Reality Editor is that multiple users may wish to collaborate to connect objects and construct programs within the same space. Previously, the system only supported a single user editing at a time, and updates a second user made would not be visible to the first unless they reopen the app. Allowing simultaneous editing by multiple users within the same network would elevate the Reality Editor into a collaborative tool where users can build upon each other’s creations.

I worked with my collaborator to rewrite a number of user interactions in a way such that they remain synchronized across multiple clients. These include creating and deleting logic nodes, creating links between nodes, and the entire Logic Crafting experience. To ensure the previously described implementation works with multiple clients, the remaining step is for the server to send new snapshots to all clients.

Whenever a client uploads state to the server, the server sends a UDP datagram to the broadcast address (255.255.255.255) which is received by every client connected to the same network. This datagram contains the ID of client who originally uploaded this snapshot, so they can ignore the update. It also contains the object address necessary to locate the resource within the client that needs to be updated. Finally, it contains the updated resource data itself as a JSON object. The client updates the indicated resource with the received values, and then renders the screen again to ensure the resources are visibly updated. In the case of a logic node, it doesn’t simply overwrite the resource with the received data because part of the state was stripped away by the sender. Instead it copies the updated state of the blocks and links, and uses these to recalculate the missing state. My contributions ensure that any time a client modifies a
block or link within the crafting board, these changes are propagated to the server and every other client as quickly as the network allows.

5.2.2.2. Invisible Node Interface

Another design decision I needed to make to rectify the front-end and the back-end involved how the data from connected IO nodes connects to the linked blocks within a logic node. Existing architecture in the server is able to propagate data from one entity to a linked resource, but the link must identify the destination by its global UID. It isn’t immediately obvious how the server will link the incoming IO node to a block within the logic node that either resides on the corresponding input cell or is linked to that cell (even if no block reside on it). It seems that the server will need to update the links in the Node View anytime a change is made in a logic node, search through the node to try to find a block connected to the input cell, and rename the link accordingly; all without knowing anything about the grid structure other than a list of blocks and links.

My solution provides a simple interface for IO nodes to connect to a logic node by encapsulating the logic with extra data structures within it. For each logic node, this involves adding a set of “port blocks” for to the server that don’t exist in the client app. These blocks have a fixed naming scheme (in0 – in3 and out0 – out3), so that when an IO node gets connected to the blue input its data can always, reliably be sent to the node’s “in0” block and the link never needs to be renamed if the contents of the Crafting Board are modified. Similarly, the data for the red output will always come from its “out3” block.

This isn’t a complete solution, because these port blocks are not yet connected to any visible blocks that the client placed. At first, it seems that there is only one step required to solve this. On the server, when a snapshot is received, recompute a set of “port links” that go from each port block to the block at the position in the grid that corresponds to their position. For instance, search for a block with position (0,0) and link the “in0” block to this block if one is found. This can be done individually for each port block, and the
data that port block receives from
the IO node will successfully reach
the block that the user expects. And
because the port blocks and links
only exist on the server, they don’t
add any complexity or edge cases
to handle within the client’s GUI.
This demonstrates the importance
of naming conventions and
separation of concerns between the
server and client.

However, there is one further
complication that arises from an
delay edge case that we need in the GUI.
We need to be able to create links
with port cells even they don’t
contain a block. If this weren’t

possible, all programs would need at least two blocks added: one on an input cell to
receive the data, and one on an output cell that receives data from the first block and
outputs it to the port. We would like to be able to draw a link directly from an empty input
cell to a single block, which can then be linked directly to an empty output cell (as with
the add block in Figure 7).

The resulting solution that fixes this problem entirely is diagrammed in Figure 20. We
include the port blocks and links on the server, as usual. But we introduce a new class
of block called “placeholder blocks” which occupy otherwise empty input and output
cells. These provide the same GUI features for linking that regular blocks provide, but
are invisible and selectively ignore touch events so that they cannot be moved or
deleted by the user. A placeholder block is deleted, however, if a regular block is placed
on its cell, because many features in the crafting board rely on the invariant that no

Figure 20. Blocks and links circled in blue only exist
on the server. Placeholder blocks exist in the server
and the app, but are invisible in the GUI.
more than one block occupies the same cell at a time. If the regular block is later removed from that cell, the placeholder block is immediately re-added to the system. The user has no idea that these invisible blocks exist other than them enabling the user to perform natural linking interactions with empty input and output cells. Placeholder blocks are synchronized with the server in the same way as regular blocks, which ensures that each port block on the server will always be linked to exactly one block from the client at all times.

The only inefficiency that I introduced with this process is that the server may need to process additional links that could theoretically be simplified. For example, in Figure 20 there are three links between “in0” and “out0” despite there being no regular blocks in between. A future researcher may wish to add a routine on the server that simplifies received logic node snapshots, for example by replacing these three links with a single link from “in0” to “out0”. This is more complicated than it might seem, because it may corrupt the state that the server sends back to clients to synchronize collaborators’ states. I found it to be of negligible consequence to leave this inefficiency in the server, but it is an area with room for progress.

The logic node implementation teaches a lesson about synchronizing state between multiple entities with conflicting priorities; the client prefers more data structures to support nuanced graphical interactions, whereas the server prefers the minimal data to process efficiently. One must either decide to give preference to one entity, or to add complexity by converting the state between the preferred formats.

5.2.2.3. Logic Blocks

The last component of the system to describe, which is arguably the most important, is the selection of logic blocks to include and how their implementation on the server actually modifies the data flowing through the system. Until this part of the implementation, the blocks that users drag from the menu into the crafting board are nothing more than empty nodes.
The internal block implementation exists entirely in the server, as it doesn’t affect the GUI at all. We created a parent directory on the server to contain a subdirectory for each type of block. The server crawls this directory to determine the full set of blocks in the system, and can query it for resources pertaining to a given block. A block implementation consists of two steps. The first is defining a set of properties, including its name, its size (how many columns it occupies), which columns have active inputs or outputs that can be linked to, and an optional set of custom data fields that can be changed in its settings page. For example, the Delay Block has a delayTime field that’s a default value of 1 second but can be changed in the settings page. The second step is implementing the render function, which decides what to write to the output when data enters the block. The render function receives a data array with up to four values (depending on the number of columns) that are each in the range of zero to one. The function is implemented by writing to the block’s processedData array and calling a callback function, which propagates the processedData to its destination.

The default block just writes each of the data values to the processedData value at the same index. The Delay Block does the same but then includes a setTimeout function that waits delayTime seconds before calling the callback. The Threshold Block, on the other hand, compares the input data to its configurable threshold parameter and writes one to the processedData if it exceeds this value, and zero if it does not. The Threshold

![Figure 21. The block icons for the blocks I implemented, including addition, multiplication, seesaw switch, IFTTT, webPost, and webListen.](image-url)
and Delay blocks were implemented by a collaborator, as were the Scale Block, Invert Block, Switch Block, and several others. I implemented a number of blocks illustrated in Figure 21, including arithmetic and conditional blocks, as well as a set of web service blocks that are described in detail in Chapter 5.2.3. I will briefly describe each of the blocks I implemented, the code for which resembles Figure 22.

The arithmetic blocks I implemented consist of Add-2, Add-3, Add-4, Subtract-2, Multiply-2, and Divide-2. Add-2 has a size of two and no custom parameters. Its render function works by inspecting its input values, and if the first and second columns have both been written to it writes their sum to the first output. There are separate blocks called Add-3 and Add-4 that do the same thing but have larger sizes and thus more inputs that can be added simultaneously. The inclusion of the Add-3 block is an innocuous example of the design decisions we made regarding the type of blocks to include. Add-3 block could be implemented in the crafting board by using two Add-2 blocks in series, which would compute the sum of \((A + B) + C\) in that order. However, we decided to include a larger set of higher-level blocks that encapsulate their functionality,

```javascript
exports.render = function (object, node, block, index, thisBlock, callback) {
  if (index === 0) {
    if(thisBlock.data[0].value > 0.5 ){
      if(thisBlock.publicData.toggle !== false) {
        thisBlock.publicData.toggle = false;
        return callback(object, node, block, index, thisBlock);
      }
    }
  } else if (index === 1) {
    if(thisBlock.data[1].value > 0.5 ){
      if(thisBlock.publicData.toggle !== true) {
        thisBlock.publicData.toggle = true;
        return callback(object, node, block, index, thisBlock);
      }
    }
  } else if (index === 2) {
    if (thisBlock.publicData.toggle === true) {
      thisBlock.processedData[2] = thisBlock.data[2];
      return callback(object, node, block, index, thisBlock);
    }
  }
};
```

Figure 22. JavaScript for the Seesaw Block's render function.
despite the redundancy, rather than force users build up desired functionality from a large combination of low-level blocks.

I also implemented the seesaw block, which is based on a physical seesaw (rocker) switch. It has a custom property called toggle, which is set to true or false depending on the values that enter the block. The first column toggles it off if that input value exceeds 0.5. Likewise, the second column toggles it on if the second input exceeds 0.5. The third column receives a stream of data, which it lets through if toggled on, or stops if toggled off. The code for this render function is in Figure 22.

The implementation of these blocks was made simple by existing architecture in the Reality Editor server. IO nodes are already processed in the same way but without additional logic in the render function. Now that the template for blocks has been implemented it is easy to introduce new blocks to the system. It has been surprising how quickly we have been able to prototype new IoT possibilities by adding blocks for different situations.

5.2.3. Web Service Implementation

Web service integration is an excellent example of using logic blocks to quickly implement powerful features. The capability to communicate data between Smarter Objects and existing web applications had been a desired feature for the Reality Editor for at least a year, but there was no front-end or back-end architecture to support it without significant effort or a clumsy methodology. Using logic blocks, I was able to implement the core functionality for this feature in a few days of work.

There are a number of reasons why users would want to connect their objects with web services. There isn't sufficient reason to implement a custom data store into the Reality Editor as a local entity, but such functionality could be outsourced to an external web service if there were a way to connect to one. Data flowing between objects could also be sent to a collective intelligence service, which can learn from the data and pass
messages back to the network for certain events. For example, if every thermostat owner connected to such a service, it could learn patterns of temperature adjustment to save money. The general problem is that objects can only be connected to other objects within the same local WiFi network, which limits the scope of what they can affect and the services they can take advantage of. Being able to pass data to and from web services would remove these limitations entirely, which is general enough that it is useful for a variety of applications, and therefore makes sense to prioritize its implementation.

Especially considering the switch to web technologies described in Chapter 4.1, it is not technologically difficult to connect to external web services. The Reality Editor has the capability to make HTTP requests. The main problem to overcome is the user experience of such an interaction; determining the most accessible way to let a non-developer connect to a web service, even if they don’t know what an API is. With the implementation of logic blocks complete, it is apparent that such a request can be encapsulated into a logic block. Data that streams into a Web Service Block would be sent to the service, and data received from the service can be streamed out of a block. To implement this, as seen in Chapter 5.2.3, the render function can include side effects in addition to writing the data to its outputs; in this example, it could post this data to a remote endpoint before processing it.

From a usability perspective, I faced considerable uncertainties about how best to let non-developers connect with their desired services, which tend to require knowledge of the endpoint URL, possibly an authentication key, and payload data formatted in a way that the receiving service can accept. When considering how to provide these, there seemed to be a tradeoff between making the experience easier for users but more restrictive on the services they connect to, vs making it more general and open to arbitrary services but require more expertise from the user. For example, a generalized system could provide fields in the block settings page where users can type the endpoint, authentication credentials, and maybe even specify how the receiving API expects the values to be formatted (for example, the name of the key that the value
should be assigned to, potentially in a nested structure). This would let a knowledgeable user send data to practically any API. A more user-friendly interface would restrict the data to be sent in a very specific, predefined format, and only support a selection of web services that were designed to accept data from the Reality Editor.

The problem with choosing either of these approaches is that they both make compromises on important aspects of the system. The user-friendly approach reduces the connectivity and capability of the system; it restricts to whom objects can connect, and by doing so, prevents them from taking advantage of the potentially powerful features of these services. The generalized, configurable approach provides these metrics, but reduces the usability and accessibility of the system because non-developers will be intimidated and likely won’t be able to use this feature. This reduces the overall trust between users and the Logic Crafting environment, which we have attempted to make simple and pleasant so that non-developers don’t refuse it out of fear of “programming”.

Excluding either extreme would be a missed opportunity, so I implemented both. Rather than choosing a single method of interaction and building one Web Service Block, I built a pair of generalized blocks for expert users called WebPost and WebListen, as well as an extensible set of blocks that make it easy for novice users to connect with common services. The example block in this set that I have implemented so far is the TriggerIFTTT Block, which connects to the relevant web platform “If This Then That”. There is still work to be done to make the expert blocks more powerful, and the accessible blocks more user-friendly, but the framework is in place for both.

5.2.3.1. WebPost

WebPost is a logic block that makes a POST request to a specified URL endpoint when data passes through it. The code for such behavior is displayed in Figure 23. This lets a Smart Object send data to arbitrary web services. The current limitations, which a future research may wish to fix, is that the received value is sent in the HTTP request body in
exports.render = function (object, node, block, index, thisBlock, callback) {
    // data flows through it like normal
    for (var key in thisBlock.data[index]) {
        thisBlock.processedData[index][key] = thisBlock.data[index][key];
    }
    // ALSO makes a post request to the server endpoint configured in publicData
    if (index === 0) {
        request.post(
            thisBlock.publicData.endpointUrl,
            { json: {blockData: thisBlock.processedData} },
            function (error, response, body) {
                if (!error && response.statusCode == 200) {
                    console.log(body);
                }
            }
        );
    }
    callback(object, node, block, index, thisBlock);
};

Figure 23. Render function for webPost block makes a post request to the configured endpoint in addition to writing the input values to processedData.

This block allows developers to build web back-ends that can receive and process data from their Smart Objects, but this format would not be accepted by most services. The current method to connect to truly arbitrary web services would be for a developer to build a back-end that accepts this specific format and can reformat it to meet the desired service’s requirements, including authentication. Example services that could be built include logging data, performing analytics on it (such as looking for patterns or anomalies), or using the data to trigger actions elsewhere on the web or on other IoT objects. I also implemented a similar block called WebPost-2 that has the same behavior except it sends two input values at the same time to the endpoint. This would be useful for logging correlated data.

5.2.3.2. WebListen

WebListen is a logic block that sets up a webhook [46] when it is added to an object. It is uniquely addressable by a fixed URL that it displays to the user in its settings page. When a value is posted to that endpoint, it outputs the value, which lets Smart Objects receive data from arbitrary web services that can send data in its expected format.
Similar to the WebPost Block, there is potential to make this even more general and accept data in any format, but research is needed to determine how to visually specify that format without adding too much complexity. This block allows developers to build web back-ends that can send data to their Smarter Objects. Example services include turning Smarter Objects on or off in response to web events such as weather data or new commits to a code repository. A promising use case that is yet to be tested is to use a combination of WebPost and WebListen blocks in separate objects in different WiFi networks to allow objects to send data beyond the current network constraints. This would open up a vast opportunity space for geographically-distributed IoT applications that has not yet been explored with the Reality Editor.

Implementing the WebListen Block required an unusual departure from solely modifying the render function. Instead, I needed to implement a new route on the server that accepts requests to WebListen block addresses, at the path '/trigger/object/:objectKey/logic/:nodeKey/block/:blockKey/'. When such a request is received, it locates the block based on the keys included in the request parameters, extracts the sent value from the body, and manually triggers the data processing engine for that block with the new value. It is regrettable to mix block-specific code into the server routes, but the potential for bidirectional web communication is promising enough to justify this block.

5.2.3.3. TriggerIFTTT

TriggerIFTTT is a logic block that provides an accessible interface for a specific web service that is useful for a wide number of users and applications. The goal is to provide functionality on the opposite side of the spectrum as WebPost and WebListen; specific, limited capabilities with a simple interface. IFTTT (If This Then That) is a third party platform that connects over 300 web services into automated “recipes” by means of triggers and effects [47]. These services include popular web applications such as Facebook, Twitter, Google Drive, and Spotify; services like SMS, phone calling, and email; and data sources like stock market data, weather data, and date and time events. They also include a number of popular IoT devices like the Nest Thermostat, Amazon
Alexa, and GE Appliances. A recipe consists of an action from one service, which when received will automatically trigger an action of another service. This means that any service which can trigger actions on IFTTT is able to interface with over 300 other services. Implementing a user-friendly IFTTT integration in the Reality Editor was thus a priority both to demonstrate how a user-friendly, service-specific interface could be incorporated as a logic block, and also as a means of eliminating the need to create individual logic blocks for each of many other popular services.

The TriggerIFTTT block interfaces with an IFTTT service called Maker Webhooks, which allows a user to trigger a participating recipe by making a POST request to a URL endpoint. A user just needs to specify the name of the recipe (the “Event Name”), and a secret key that IFTTT provides the recipe creator (the “IFTTT Key”). The TriggerIFTTT block settings view contains text fields where both of these strings can be entered. The block then becomes fully functional, and will trigger the designated recipe anytime it receives data. Providing the Event Name and IFTTT Key are simple enough tasks that a user will be able to configure the block without programming experience or even knowledge about web requests. The implementation resides entirely within the render function, which generates the correct endpoint URL using the provided event name, and authenticates the request using the provided key. A similar model can be used to implement clean interfaces for other authenticated services in the future.

5.2.4. Locking Implementation

A side effect of leaving web service connections in one’s object network is that some of those blocks may have sensitive authentication data saved in them, which any other user could browse with the Reality Editor. This is one example of a growing need to consider data privacy and security in this system design. These issues become more important as the Reality Editor becomes more powerful in the types of things it can connect, and more prevalent in the number of users who might interact with the same set of objects.
Figure 24. Nodes can be locked or unlocked with the buttons in the lower right. Locked nodes are semitransparent and cannot be opened or modified.

The current system provides unrestricted read-write access to all users over all objects in the system, which may be a concern if the system is storing sensitive data or if an improper configuration of the network may result in destructive behavior of the physical objects it contains. To mitigate these concerns, I implemented a permissions control system that allows users to restrict access to certain objects and links to users with sufficient permissions. With the press of a button, a user can lock a node to prevent others from modifying or deleting it or its set of links. A link with either connected node locked cannot be deleted, and new links can only be drawn between nodes if both nodes are unlocked. Subtle visual feedback shows users which nodes are locked or unlocked, as seen in Figure 24.

I prototyped two distinct authentication methods for modifying locks. The first is a fingerprint-based method that authenticates the owner of a device using its native TouchID SDK, and locks the resource with the UUID of the device. This lets the owner of the device enter Locking Mode with the scan of a fingerprint, after which they can
lock or unlock any of their resources. The fingerprint is a clean, minimal user interaction. However, there is no way for multiple users to have permission to adjust the same lock. And if the user loses their device, even temporarily, any locks created by it will be impossible to remove until the device is reacquired.

The second authentication method combats these problems by using a passcode. To enter Locking Mode, the user first authenticates a passcode that will be used to lock or unlock resources during that session. In this manner, the user can lock resources with different passcodes that can be shared with other users or kept private. This mitigates the risk of having a single point of failure for the lock authentication (the user’s phone). However, it introduces problems of its own, which must be considered. A prominent one is that a user must now remember at least one password, and forgetting it will result in a permanent lock with no recovery mechanism. The second problem is in the visualization: if a user locks a dozen resources, each with a unique passcode, it will be hard to remember which passcode is associated with a given object. The fingerprint method guarantees that all of a user’s locks can be unlocked simultaneously with no risk of forgetting. The passcode method is used in the final implementation, but further work might discover how to improve this interaction.

Once a user enters locking mode, three additional buttons appear to lock or unlock nodes on the screen. The first is the Lock Button, which takes all visible, unlocked nodes and locks them under the current user’s authentication. This prevents all users (even the authenticated user) from modifying the node or its links until it is unlocked, which is useful to prevent accidental changes to a delicately constructed network of objects. There is also a Half Lock Button, which takes all visible, unlocked nodes and places them under ownership of the current user, without restricting users from modifying them. It effectively “locks them open” instead of locking them closed. This is useful if a user wants to claim a node as their own so that they can continue to modify it in the future without risk of another user locking it, but they also don’t want to restrict other users’ access to it. The final button is the Unlock Button, which completely unlocks any visible nodes whose locks the user controls.
I considered many different schemes for how to control access to nodes, including having more levels of restriction that would allow or prevent subsets of actions. I also considered different forms of identity and authentication that would allow a user to belong to a group of authorized users. In the end, I went with the simpler, three button mechanism to reduce the cognitive load on the user and keep the user interface simple, as the feature shouldn’t occupy a disproportionate amount of screen space compared to its utility. Future researchers might wish to explore how to provide more flexibility in the user permissions model while minimizing the negative effects on the complexity of the user interface and its cognitive model. Future work is also needed to better visualize which nodes are locked by which users or passcodes. Currently all locked nodes look the same, regardless of whether the current user has ownership over their locks, which is confusing when trying to predict whether tapping the Unlock Button will be successful on a desired resource.

A final recommendation for future work requires some concluding remarks on security and privacy. Regarding privacy, it is important to note that all data storage takes place locally within the user’s private network. The only time a user’s data may ever be seen by a third party is when the user chooses to connect the data to a web service in exchange for the value it provides. These connections are displayed visually, ensuring transparency regarding the privacy of his or her data. In short, the strong privacy of the Reality Editor is one of its positive aspects, especially in an era where that transparency is so rare. The same claim cannot be made yet about security. It is important to note that the locking mechanism is not really the “security” feature it might appear to be. There have been no attempts so far to provide formal, cryptographic security of the locks, so a malicious user could easily bypass them to gain access to a locked object. The current locking is better described as a usability feature; it is primarily intended to prevent other users from accidentally changing objects that have been carefully configured. It could be used, for example, by a parent who does not want a child to delete the links and programs they built into their home automation system, but still wants to let them play with the light colors in their bedroom.
6. Prototypes

6.1. Augmented Reality Content Authoring and Browsing

We demonstrate the breadth and depth of the capabilities the Reality Editor offers as an AR platform for authoring and browsing content. The switch to web technologies opened a wide possibility space of applications. The simplest category of applications take advantage of the fact that any existing web content can be embedded and rendered in the interfaces. There is an entire web browser within the Reality Editor, so any webpage can be included in an interface as an iFrame; for example, a Youtube video or a Wikipedia article can be used as AR content with a single line of HTML (as in Figure 25). We explored a number of more powerful use cases. The first, 6.1.1, shows a simple artistic example of using AR to see otherwise invisible effects. The next, 6.1.2, makes use of WebGL to render immersive 3D content into space, to demonstrate that the Reality Editor can compete with existing game-oriented AR platforms in terms of the production value that can be included in the experiences. 6.1.3 demonstrates how the Hybrid Object API can be used to build AR experiences that are responsive to their spatial environments a way that other AR browsers would have trouble implementing. 6.1.4 shows how easy it is to connect these AR experiences to existing web back-ends. A bigger example is presented in 6.1.5, which shows how an addition to the Hybrid Object API makes it possible to create more seamless, customized experiences between the Reality Editor and the displayed interfaces. It also shows how to include more behavior for a single object. Finally, in 6.1.6 we extend the Spatial Search view from 6.1.5 to test for scalability in a real world environment.

Figure 25. Augmenting a physical object with Youtube (left) or Wikipedia content (right).
6.1.1. Augmented Art Gallery

I've already discussed augmenting an object with existing web content, but we can build new web content specifically to be viewed in AR. A technically easy, but surprisingly effective, demonstration of this is augmenting an art gallery with additional visuals for each piece of art. At a traditional art gallery, visitors only see the finished work of art, which may be beautiful in itself but can be hard to connect with unless there is additional information displayed. Usually a placard will display the title, artist, and sometimes a description of the work. But, with the exception of some high-end museum exhibits with collections of sketches and related media, visitors rarely get to see the highly visual, creative process that went into the work itself. The Reality Editor makes it easy to display such augmented visual information to visitors and place it into space within the gallery. A webpage can easily be authored that displays a sketch or a time-lapse video of the creation of an artwork; any developer who knows how to display an image or embed a video in HTML could do so. These interfaces can then be positioned adjacent to, or in front of, the artwork in the gallery using the Unconstrained Repositioning tool. Visitors can then point the Reality Editor at art in the gallery to see its creation story come to life before their eyes, as seen in Figure 26. This lets visitors connect to the artwork on a deeper level without any physical installations that the gallery needs to pay for or plan into the arrangement of the space. This showcases the

Figure 26. Two examples of using AR to view additional information at an art gallery. The Reality Editor can show a sketch (left), or float a time-lapse video (right) alongside the art.
AR browser as a tool for forming deeper connections with our physical environments by displaying information that would otherwise be logistically prohibitive to make visible.

6.1.2. Immersive 3D AR

Next I made some example interfaces to show just how much we could achieve using web tools as the foundation for our rendering. Some users might be surprised by the choice of web technologies, since web is typically consumed as 2D content such as the example in 6.1.1. AR, however, connotes graphically-rich 3D overlays, with the exception of niche systems that render 2D windows over the environment [48]. While the Reality Editor similarly has a preference for 2D content, it would be a weakness if it could not support 3D interfaces. This would argue in favor of using a game engine like Unity rather than the web, since game engines are built for rendering complex, visually appealing and entertaining 3D content. To contradict this assumption about the web’s inability to render 3D content in AR, I prototyped a number of demos using the Three.js web framework [49].

Figure 27. 3D animated model rendered in WebGL using the Three.js framework.
The simplest Three.js demo renders a cube on top of an interface. The HTML for this interface is included in Figure 21. This has only a few steps. The first is to initialize the Three.js camera, scene, and renderer, and to place the cube at the origin. Next, the Hybrid Object API is used to disable default transformations by making the interface fullscreen, and then add a listener that receives the modelView and projection matrices each time the Reality Editor renders a new frame. The reason they need to listen for these matrices is for the correct rendering of the 3D content. These scenes handle their 3D transformations internally, rather than having the Reality Editor transform their iFrames externally.
In addition to the simple cube, I built two more demos that make use of the Three.js framework. The first extends the graphics to render an animated turtle model rather than a simple cube, as in Figure 27. The next is a fun but intriguing demonstration of linking the physical and virtual worlds. It includes a teapot model that can be stretched or warped in two ways. It is connected to a the rotation IO node of a physical knob that can be rotated between 0 and 100%. The values outputted from this knob are sent to two IO nodes that reside on the teapot. Rotating the knob increases these values which stretches and distorts the virtual teapot model. This is a persuasive demo of the power of AR for media consumption as well as connecting a purely digital interface with the physical world.

6.1.3. Context-aware AR

In addition to building entertaining media interfaces, it is also important to explore more practical examples that are applicable to everyday life. In this vein, we collaborated with Target to prototype AR interfaces to improve the in-store experience for shoppers.

One successful demo displays nutrition comparisons between multiple products that are visible at the same time, as seen in Figure 29. This makes use of a few parts of our Hybrid Object API. It detects when objects become visible or invisible. When these events occur, it broadcasts global messages to all other visible objects. Upon receiving a global message (which contains the sending object's nutrition information as a JSON object), the receiving interface will update to show which nutrition facts are better or worse.

Figure 29. A context-aware AR application that compares nutrition information of grocery products viewed simultaneously with the Reality Editor.
worse than the other visible products. This demo shows how we can build context-aware AR content. It also demonstrates a convincing use case where AR helps users digest a large amount of information on demand by presenting it in a clean visualization that would otherwise take a long time to understand.

6.1.4. Back-end connected AR

As mentioned in Chapter 4.1, a benefit that cannot be overlooked when using web as our platform for AR is that users or organizations with existing back-ends implemented to serve data/content to their websites can use those very same back-ends to serve their AR interfaces. I demonstrate this in collaboration with Target, who has an existing API called the Item API that contains information about all of their products. For grocery products, this includes which certifications the food has, including whether it is Organic Certified. By providing API key for this product in the interface, it can make an HTTP request to the back-end when the interface is loaded and retrieve the corresponding product information. The proof of concept prototype has interfaces set up to augment a

Figure 30. Product information visualization using live data from the Target Item API.
number of cracker boxes. When the user points the Reality Editor at a cracker box, it
will fetch the data from the API and then display one of two icons to indicate whether
that product is organic, as depicted in Figure 30. This is a simple example of using live
data in an interface without needing to change the back-end at all. This is the only demo
where live data was used in this way, but the user interaction for the prototype is
expanded in Chapters 6.1.5 and 6.1.6.

6.1.5. Menu-reactive AR

My collaborator implemented a new architecture for the sidebar, which tracks the state
of the Reality Editor (which mode it is in, and which view is showing), and switches the
buttons when this context changes. This supports a new Retail Mode that can be
enabled in the settings page, which swaps out the buttons in the Node View for a set of
category-specific buttons that are useful for browsing AR experiences in a retail
environment. This menu can be seen on the right side of the screens in Figure 32. My

```html
<body>
  <div id="gui"></div>
  <div id="info"></div>
  <div id="tag"></div>
  <div id="search"></div>
  <div id="work"></div>
  <script>
    var viewNames = ['gui', 'info', 'tag', 'search', 'work'];
    var obj = new HybridObject();

    // callback gets triggered whenever the state of the Reality Editor changes
    obj.addInterfaceListener(displayView);

    // hides all views except the view selected from the realityEditor
    function displayView(selectedInterface) {
      viewNames.forEach(function(viewName) {
        if (viewName === selectedInterface) {
          document.getElementById(viewName).style.display = 'inline';
        } else {
          document.getElementById(viewName).style.display = 'none';
        }
      });
    }
  </script>
</body>
```

Figure 31. HTML template for building a menu-reactive AR interface.
collaborator also implemented a new function for the Hybrid Object API that lets interfaces subscribe to these state updates about the Reality Editor. It triggers a callback on the interface whenever a sidebar button is pressed that toggles the interface state. For example, all visible interfaces with this subscription would be notified when the Reality Editor switches between the Node View and the Crafting View. They would also be notified when switching between any of the five buttons in the Retail Mode. This forms the basis for this prototype, which is called a menu-reactive AR interface.

This menu-reactive AR interface follows the theme of 6.1.3 and displays additional information about a retail environment, specifically in a Target store. The use case scenario is that there is a customer in a grocery aisle of a Target store who wants to learn more about the products before deciding which to buy. The interface distributes different parts of this information across views that can be toggled between using the Retail Mode buttons in the Reality Editor. Figure 31 provides a template for an interface that subscribes to the state changes of the Reality Editor with the addInterfaceListener

Figure 32. When different sidebar buttons are pressed, it shows product information (upper left), coupons (upper right), nutrition (lower left), or spatial search (lower right).
function of the Hybrid Object API, and displays a corresponding div for each mode. The five views that can be toggled between include: product video and reviews, nutrition comparison, coupons, inventory heatmap, and spatial search.

Product video and reviews demonstrate the ability to help customers make better informed decisions by embedding existing web content into the in-store environment. This information would typically only be available online, but can now be viewed within the spatial context of the store. The nutrition comparison view simply displays the context-aware AR prototype from Chapter 6.1.2, again showing the benefit of AR’s ability to scan multiple products at once. The coupons view shows how particular content can incentivize AR usage, by revealing discounts at a glance that would otherwise be invisible. Also of note is that because the interfaces are built with web technologies, it is easy to connect this view to the store’s web back-end to continuously provide up-to-date coupons. This is also easier than individually scanning barcodes to see if there are any discounts. Instead, an entire grocery aisle can be examined at once to reveal all coupons and discounts, saving time for the user. The inventory heatmap view is intended for employees as they restock shelves; it indicates how many units are left in stock. The spatial search view allows the user to select preference about which types of products they are interested in, and see at a glance which products meet those criteria. In our demo, you can filter things such as allergies (dairy, gluten, nuts) societal preferences (organic) and health preferences (sugar, corn syrup).

Together, these views show how a menu-reactive AR interface can provide a rich suite of information to users in a real life situation. Future work should explore how to provide more sets of menus that can dynamically switch based on the user’s context, rather than needing to manually program them into the editor and enable them in the settings page.
6.1.6. Industry-scalable Spatial Search

To evaluate the scalability of the system I created a demo that includes an order-of-magnitude more objects than the Reality Editor had ever dealt with before. It is a scaled-up version of the spatial search demo described in 6.1.5, which operates on a set of 200 types of cereal boxes found in a grocery store aisle. A working demo is pictured in Figure 33. By building this demo, I discovered a bottleneck in the object creation process. Despite the improved developer usability that web provides, the reliance on a third-party, closed source SDK for the image recognition (the Vuforia SDK) means that the Reality Editor cannot optimize the marker creation process when generating new objects. All other parts of the process could be automated, but this bottleneck forced me to manually generate each of the 200 markers for objects.

A second issue that I discovered is in the object downloader process in the native iOS app, which discovers objects via UDP broadcasting and subsequently downloads the resources for each. Typically working with a dozen objects at most, this process was

Figure 33. Using preferences to filter and display relevant products, with hundreds of products loaded into the system.
never optimized for industry-scale sets of objects. Upon opening the app, this process took an average of three to five minutes of continuous downloading to obtain the necessary resources for the demo. This is prohibitive for deploying the system in real world applications such as a grocery store, despite having a compelling use case. Future researchers should investigate optimizing this process or using a different technique to acquire the interfaces for each object.

6.2. Logic Crafting

6.2.1. Lego Playground

We evaluated the Logic Crafting interface via a suite of applications built with Lego WeDo 2.0 bricks, which follow in the tradition of Lego Mindstorms in letting children build reprogrammable robots with legos, sensors, and motors for playful and educational purposes [50]. The Reality Editor connects to the bricks using bluetooth, and can send and receive data to their connected lights, distance sensors, motors, and buttons. Their modular nature and creative, playful connotations provides us with the perfect playground to experiment with the Logic Crafting. Having a working environment

Figure 34. Node View showing a Lego WeDo with a distance sensor, motor, button, and light.
for testing also inspired the construction of new blocks for the system to provide missing functionality.

These prototypes used five sets of WeDos, each with a sensor and a motor, which were set up to match the one seen in Figure 34. These let us imagine various home automation scenarios that would involve sensors and actuator, and program the logic using the lego sensor and motor as a proof of concept. We also used the WeDos as toys that could be reprogrammed with AR. For example, we built a Lego car and racetrack, as in Figure 35. When a button on the car's WeDo was pressed, a sequence of red, yellow, and green lights would light up with a delay between each, then the car motor turns on and it drives forward until it reaches the finish line, which is detected by a distance sensor on the sidelines. At this point a toy flag waves around to mark the end of the race, and the car continues to drive until the distance sensor on its front detects that it is about to run into a wall. At this point the car stops and the program is complete. The complexity of this use case is a proof of concept that our Logic Crafting system can in fact be used to construct useful programs that capture a user's intended behavior.

We were also able to prototype a number of simple programs to evaluate the range of possibilities that our system enables. A collection of these programs are illustrated and explained in Chapters 6.2.1.1 through 6.2.1.7, to give concrete examples of the logical concepts that this programming environment can capture. A set of Lego WeDos is an easy, accessible set of hardware with which to test these programs, but they could in theory be connected to any type of sensor or actuator instead of these modular toys.

Including the WeDos in the explanation of these examples is meant to make them concrete, but they are not limited to a particular type of hardware.

Figure 35. This Lego car uses Logic Crafting to drive forward until the distance sensor on its front hits a wall.
6.2.1.1. Start and Stop

The program in Figure 36 uses a delay block and an inverter to trigger an action, and then stop it a few seconds later. As a test scenario, the green input is connected to a button and the green output to a light. When the button is pressed, a 1 arrives in the green input and immediately turns the light on because of the link from the green input cell to the output cell. But it also sends a 1 to the delay block, which will wait a few seconds and then send the 1 to the inverter block. This changes the 1 to a 0 and sends it to the green output, which turns the light back off. The amount of time the light stays on can be changed by tapping on the delay block and choosing a number of seconds.

6.2.1.2. Delay Chain

The program in Figure 37 lets a single action trigger a sequence of events with delays in between each one. As a test scenario, the blue input is connected to a button, and lights are connected to each of the outputs. This will immediately turn on the first light (because of the direct link from the blue input cell to the blue output. It also triggers the delay in the green column, which will wait a few seconds before activating the light connected to the green output. It also triggers the yellow delay block, which does the same for the red, eventually turning on all the lights.
6.2.1.3. Arithmetic Function

The program in Figure 38 uses a scale, add, and multiply block to compute a mathematical function on the input signals and output that value to blue. As a test scenario, the blue, green, and yellow inputs are connected to distance sensors, the blue output is connected to a light, and the scale block has a scale factor of 3. In this case, the blocks compute the function:

\[
\text{blueOut} = f(\text{blueIn}, \text{greenIn}, \text{yellowIn}) = \text{blueIn} \times (\text{greenIn} + (3 \times \text{yellowIn}))
\]

Chaining together arithmetic blocks can be used to compute a variety of functions on the input data. In the future, more mathematical operators could be added as blocks to build even more functions.

6.2.1.4. If Both Are On (Logical AND)

The program is Figure 39 an interesting example of building a basic operator for formal (digital) logic that works on an analog stream of data. The threshold blocks act as analog-to-digital converters [51] which output a 1 or 0 depending on the amplitude of the signal. The multiplication block expresses the same truth table as the logical AND operation because of the properties of Boolean arithmetic [52], so passing the digitized signals through this block results in a logic node that acts as an AND gate.

This program is immediately useful in a lot of situations where a user wants to know if two objects are both "on" (meaning they have values of 1, or at least something above a
certain threshold). It is also useful in a more theoretical sense because of the functional completeness of NAND gates [53]. This means that any Boolean function can be built up from a set of negative-AND gates. Logic crafting already has an inverter block, which could be combined with the output of the above example to produce a NAND logic node. This offers a proof by construction that any Boolean function can be computed by logic crafting. A computer processor could then, in theory, be constructed out of logic nodes. This may be an educational opportunity to use the Reality Editor and a set of sensors and lights to teach Boolean logic and visualize the results in a spatially-constructed environment of sensors and lights.

6.2.1.5. If Either Are On (Logical OR)

The program in Figure 40 works almost exactly as the Logical AND example above. The only difference is that we use an add block instead of a multiply block. This computes the Boolean OR function, because the only time a 0 is outputted is when both digitized signals are at 0. Otherwise, the sum will be 1 (if only one of them is on) or 2 (if both are on). The add block has a floor and ceiling on the output that keeps it within the range of 0 to 1, so the output will be 1 even if both are on.

This example has the same (redundant) theoretical implications as 6.2.1.4. A more practical example for this would be to connect two lights to the inputs, and a radio to the output. This would make the radio play when either light is on, but when they are both off it would stop playing.
6.2.1.6. Rotating Car

The program in Figure 41 shows a more domain-specific logic node that is intended for a Lego WeDo car with two wheels. If both rotate forwards, the car will drive forwards. If they both rotate backwards, the car will drive backwards. But if they spin in opposite directions, the car will turn.

This program uses a sensor connected to the blue input to control how fast the car spins. Each motor is connected to a different output. The full range block on the blue input cell maps the sensor values from the range of $[0,1]$ to the range of $[-1,1]$ so that it can make the motors spin forwards or backwards rather than just forwards.

The link to the blue output makes that motor spin in one direction, while the other link that passes through an invert motor block makes other motor spin in the opposite direction at the same speed. This example demonstrates how including high-level blocks for specific use cases, such as Lego motors, reduces the complexity of programs, which in turn makes the tool more accessible for children and hobbyists.

6.2.1.7. Engine On/Off Switch

The program in Figure 42 is another domain-specific logic node, intended for a Lego car, although the core logic could be adapted for other scenarios. The program uses a seesaw block to allow or stop the data flowing from the yellow input to output. In this scenario, let the yellow output be connected to
the motor, let the yellow input be connected to a sensor that sets the speed, and let the blue and green buttons be connected to buttons.

Pressing the green button activates the seesaw block to "turn on the engine" by allowing data to flow into the motor from the full range block. While it is on, the speed of the motor will be updated by the sensor. Pressing the blue button "turns off the engine" by sending a 0 to the yellow output (with the lower left link), and toggling the seesaw block off, which stops new speed values from passing through.

6.2.2. Web Services

Each web service block was evaluated with a small demo. To test the WebPost block I implemented a Node.js server that was run locally, which has a route configured that receives data in the correct format for WebPost. This server used the "http" node module to create the server, as demonstrated in Figure 43. It listens for POST requests with a certain URL, parses the body, and inspects the body's blockData, which is an array. In this case, the array only has one value, because the webPost block is only one column wide and hence only sends a single value per trigger. A similar API can be used with the WebPost-2 block, however, which sends two values at a time, so the block data array would contain both values. This example server just stores received values to an

```javascript
var http = require('http');
var receivedValues = [];

http.createServer(function(request, response) {
  if (request.method == 'POST' && request.url == '/test') {
    var body = [];
    request.on('data', function(chunk) {
      body.push(chunk);
    }).on('end', function() {
      body = Buffer.concat(body).toString();
      var value = JSON.parse(body).blockData[0].value;
      console.log("Received a " + value);
      receivedValues.push(value);
      response.end("{}");
    });
  }
}).listen(8082);
```

Figure 43. Code for a Node.js server (right) that can receive and log data from a webPost block (left).
array, but it could do anything with the data ranging from running analytics on it, to visualizing it, to forwarding it to other local or remote services.

I primarily tested the webListen block within the terminal using the curl command to make HTTP requests without building a front-end, as demonstrated in Figure 44. I tested this by connecting the output of the webListen block to a light, which I was able to turn on or off by running the command with different data attached in the body. Future work would involve evaluating the utility of this block more thoroughly by building a front-end web application running outside of the local network, from which users can trigger different actions on a set of objects.

I began implementing a larger demo to evaluate how these blocks can be used for bidirectional remote communication between Smarter Objects on a large scale, as mentioned in Chapter 5.2.3.2. This involves hosting a server at a static IP address that can be given to the webPost block. Upon receiving data from multiple webPost blocks from different users (and thus different IPs), the server can maintain a table of known client objects and their IP addresses. When one Smarter Object wishes to communicate with a geographically distant object, it can send the data to this server, which will look up the IP address of the intended recipient and forward the data to a webListen block on that client object. The overall idea is promising, but it is yet to be implemented because of many architectural details that have not been explored. For example, it is unclear what the best way is for the server to maintain an up-to-date mapping of IP addresses to webListen blocks. Future researchers may also wish to investigate the best user

```bash
curl -X POST http://192.168.1.12:8080/
  trigger/object/lego40i73qtpsc80n/logic/QDdc6325x66w/block/blockUc13ipum7vr/ -d
  '{"value":1}' -H "Content-Type:
  application/json; charset=UTF-8"
```

Figure 44. Terminal command (right) to send a value of 1 to a particular webListen block (left).
One set of demos that has been more clearly implemented are the IFTTT web service connections. I implemented several demos that use this block to trigger different actions online. The first logs all data sent to the block into a Google Sheets document. I connected the output of a Lego WeDo distance sensor to an IFTTT block set to trigger such an event. The result is a live-updating data log that a user can later browse and perform calculations on. An excerpt is included in Figure 45.

I constructed another IFTTT prototype that connects to a mobile push notification service called Pushover [54], as seen in Figure 46. I created a recipe with a Maker Webhooks trigger that listens for an event named "button", which gets called by the TriggerIFTTT block when it receives data. To test this, it is connected to a physical button, which can be pressed to send a push notification to the device associated with the connected Pushover account. This is a trivial use case, but the technology is powerful. It could be used, for example, to receive a notification whenever a user's doorbell rings, or their oven finishes cooking, or their baby monitor picks up a loud sound, even if those pieces of hardware don't include a feature for push notifications by default. The push notification prototype, while simple and abstract, is demonstrative of how the TriggerIFTTT block allows a user to combine any hardware functionality with
any common web service features. A future researcher may wish to further enumerate useful combinations made possible by this capability.

Figure 46. The IFTTT recipe (left) connected to an object via a TriggerIFTTT's settings page (upper right) triggers a push notification when the button is pressed (lower right).
7. Conclusions

As introduced in Chapter 1, the contributions of this thesis were motivated by the desire to answer two central questions. The first involves allowing people to control and interact with IoT objects in a more meaningful way than previously possible, so that these objects are aware of and customized for the user’s specific context and needs. The second involves making augmented reality content more pervasive and powerful.

The best way to accomplish these goals was to extend the features of an existing tool that had already begun to solve both of these problems. The Reality Editor let users interact with Smarter Objects via AR user interfaces, and to connect their functionality on demand into simple causal relationships. I worked with a team to extend the Reality Editor by building up a comprehensive ecosystem for building, deploying, browsing, and customizing bidirectional AR content.

The motivations, combined with the existing functionality of the Reality Editor, led to four specific questions that I pursued, as asked in Chapter 3. Through the implementation of a web-based AR browser, the Logic Crafting visual programming environment, and some auxiliary tools for data manipulation, communication, and access control, I was able to answer each of these questions. Conclusions drawn about each question are described in 7.1.1 through 7.1.4.

7.1. Question 1: AR Pervasiveness

How can we make AR content a more ubiquitous part of life? How can it become an additional virtual layer with which the entire world can be interacted? Specifically, how can we improve the content creation process such that it becomes easy for developers to deploy meaningful AR experiences into the world?

A key insight to building a platform with ubiquitous potential was the realization that we did not need to reinvent anything. The problem had already been solved for the Web
decades ago with the advent of HTML, CSS, JavaScript [55], and the authoring tools and frameworks that arose from this foundation. Rather than invent a new format and associated tools for building interfaces, as the preexisting Reality Editor briefly pursued, it makes sense to give the Reality Editor the ability to render webpages in 3D space. After an upfront time investment to shift the platform to web, we can take advantage of everything the web offers without any additional developments. While we could have rendered our interfaces with a number of technologies, the web's robust developer ecosystem and time-tested scalability is exactly what the Reality Editor needed. It has proved useful when building prototypes, as the available frameworks and built-in capabilities for networking have made it possible to rapidly create useful experiences. Combined with the repositioning tool for placing these creations into space without prohibitive expertise, it becomes a much more accessible platform for deploying content. The hope is that the ease of the platform will improve developer adoption, who will create useful content that will solve the chicken-and-egg problem of AR [56].

7.2. Question 2: Intelligent Network Customization

*How can we design and engineer a system that allows users to build up useful, intelligent, customized networks from less intelligent objects in an accessible and extensible way?*

With Logic Crafting, we introduced a VPL that combines the ease of use of languages like Scratch with a data flow structure like Lab View that is better suited to the data passing between connected IoT objects. Learning from the criticisms of existing data flow VPLs, we paid special attention to the user interface and in providing high-level blocks that let users put their ideas into space without requiring too many blocks and links. By containing this programming environment in logic nodes and composing them out of logic blocks, we achieve a layer of abstraction between the blocks and nodes. Blocks can be combined into functional groups within a single logic node, which hence can be treated as a black box in the Node View. This is helpful for managing the complexity of customized networks of objects which might have dozens of logical
operators, as the end user can view the flow of data at a coarser granularity unless actively editing a specific logic node. This is key for an accessible system designed for non-developers, as an increased cognitive load would quickly overwhelm their understanding of the system and reduce adoption. Complexity can be added to a logic node without affecting the rest of the system. Modularizing functionality into logic nodes will also be useful in the future if we wish to give users the ability to compose these nodes into even larger modules, or for the ability to duplicate or share nodes with other users.

Another key insight came in understanding the tradeoff between usability and completeness. As an engineer, it is tempting to desire overly-generalized systems that give a formally complete set of functionality. But unless the system has an inherent, elegant simplicity to its completeness this often results in adding a lot of complexity. In some cases this is fine, such as building a tool for engineers that value capability above all else. But complexity often conflicts with usability, due to the necessary user interface components and the added cognitive load. Logic Crafting, as it is currently implemented, has not been designed with Turing-completeness [57] in mind, and has obvious gaps in its features. There exist programs that cannot be built in Logic Crafting, and not every block or feature that we conceived was eventually added to the system. Logic Crafting is meant for the average user to be able to give their objects simple instructions about how to interact. The majority of use cases will only require a minority of the features, by virtue of the power law [58]. When building a system with accessibility and usability at its core, this reality means deciding a utility threshold where you start excluding promising features. The result, which may be disappointing to engineers, is a non-complete system. But the benefit is a system that is easy to use and supports the most common situations.

7.3. Question 3: Collaborative Environments

*How can we encourage and facilitate a collaborative editing environment where users can come together to augment their shared spaces for the collective benefit?*
Editing a shared space has positive and negative consequences that will become ever more prominent as the system adoption increases. Currently most of our prototypes have taken place with one to three users and fewer than two dozen Smarter Objects. If the work of this thesis contributes to increasing the prevalence of such AR-enabled IoT networks, one could expect hundreds of Reality Editor users to pass through the same spaces with perhaps hundreds of connected objects, for example at a school, office, or dormitory. Hundreds of users can contribute intelligence to the shared environment in the form of new logic nodes and links, and they can see each other's changes update live. This allows for a collaborative platform where users can edit, remix, and learn from one another's creations.

However, there are risks associated with such a collaborative environment that must be considered with even greater weight. Users may have conflicting needs and preferences for how the space should behave. They may edit or delete other users' creations to replace them with their own. They may accidentally or maliciously delete links or adjust other settings. This is a serious problem because a user who believes their physical space is configured one way may make uncorrectable, perhaps even dangerous, mistakes if their configurations have been altered without their knowledge. Severe but very possible examples are not hard to imagine; for example, connecting the a bedside lamp to turn on the oven could start a fire in the middle of the night. The locking mechanism begins to solve this problem by letting users control access to their objects. However, it requires an active approach; resources are unrestricted by default and only become protected when a user remembers to lock them. This leaves room for mistakes that ideally would be prevented.

Features cannot be added recklessly without considering the negative consequences. Currently, the locking mechanism serves as a temporary solution to mitigate the problems that arise from conflicting users sharing the same objects. However, a more comprehensive design is needed for shared spaces to not detract from the system at all. An idea for such a solution is the ability to save a snapshot of the current network
configuration: which IO and logic nodes exist and how they are linked together. Users could backup their preferred version of the network, overwrite the current network with theirs when they are in the space, and allow others to overwrite their own when they are not in the space. Even better would be a system that automatically detects who is in the space and adapts the network structure based on their saved snapshots. For example, the kitchen might adapt to preferred appliance settings and lighting ambiance when one family member is in the room, but have entirely different settings when a different member is present instead. When they are both in the room, it could either interpolate the settings to find the best compromise for the two, or ask for confirmation before using a particular setting. For example, the lights might adjust automatically to the average brightness of the two users, but the toaster would ask for confirmation before using the first user's settings (perhaps lightly toasted) or the second user's (dark and crispy). In the case of the toaster, it would be suboptimal to average the settings because there is no need to compromise when it is activated individually for each user. The confirmation is preferable, but even better would be a method of automatically detecting which user is operating the physical device so that the space behaves as each expects without additional input.

7.4. Question 4: Building Trust

How can we build tools that ensure users’ data privacy and security and build their trust in the shared/connected system/network such that they actually use it?

The locking tool also helps users with their data privacy; for example, a locked logic node with a user’s web service authentication key can be locked so that other users can’t view it. But more than any tool implemented in this thesis, data privacy is mainly addressed by the inherent transparency about where data is flowing. Everything is represented by a visible link that is legibly routed from its source to its destination so that a user can tell exactly whom is receiving what data. The fact that the system primitives (nodes and links) serve this purpose better than any added features gives an insight about system design, and the importance of solid foundations. Systems form
often unexpected emergent properties based on the interactions between their primitive components [59] For the Reality Editor, this includes privacy features.

Another key insight about the data transparency ensured by the Reality Editor relates to the lack of such transparency in other technologies. Online platforms, for example, are notorious for selling user data for the purpose of targeted advertising [60], but there is little visibility as to when, where, and to whom this is happening. Now imagine visiting Facebook or Google and seeing visible links from one's profile data and activity history to every client to whom these data is being sold. It would likely be an uncomfortable sight that would deter users from participating in such platforms, unless they felt the value they gain is worth the links their data flows into. This radical transparency [61] would allow users to make informed decisions about the platforms they make use of, and hence increase trust between the users and platforms they remain a part of. Now imagine an even more radical reality; users who are uncomfortable with the destination of any of their data could swipe through links to prevent that flow of data. This may seem utopian, but it is exactly the level of transparency that the Reality Editor provides. The only time data can possibly leave a user's local network is when they draw a link to a web service, which is prominently visualized and can be deleted anytime the user feels that the service is not providing adequate value. This makes a statement about the Reality Editor and also about the world we could live in. Such transparency may never permeate established corporations, but perhaps users who realize the value of such a system may grow to expect the same in other areas of their virtual lives.

A final insight came in deciding what not to implement. I did not implement persistent data stores as a first-class citizen, despite spending significant time researching the possibility, because the interface would be overly complex and the functionality can be largely accomplished by web services instead. I also did not attempt to implement rigorous security with the locking system. It was not within the scope of the thesis to engineer a system that would guarantee any reasonable level of security in a fully-realized system deployed in the wild. Rather than halfway-implement security features that add complexity while failing to accomplish anything substantial, it is better to leave
the locks as a deterrent and as an error-prevention mechanism. Creativity means being willing to entertain any proposed feature, but preserving simplicity and usability means being skeptical of each. This duality was seen in full force while developing the Reality Editor.

7.5. Summary of Contributions

In summary, my specific contributions to the Reality Editor include contributions to the AR authoring platform, Logic Crafting, web service integrations, locking, and in the construction of numerous prototypes, as outlined in Chapter 1.2. For the AR authoring platform, I built the communication channel between JavaScript and C++ that allowed the platform to make the important shift to web technologies. For Logic Crafting, I assisted with paper prototyping the preliminary interactions, implemented the entirety of the front-end, implemented some of the back-end, and created several logic blocks. I also fully designed and implemented the web service integrations and locking mechanisms.

More than features added to an app, my contributions have design and engineering implications for the wider fields of augmented reality, the Internet of Things, and data flow VPLs. One of these contributions is helping to define, design, and implement a grid-based data flow VPL within an AR application. Design decisions regarding the grid positioning constraints, the abstraction level of the blocks, the routing algorithm, the touch interaction model, and the preservation of context between the crafting board and the containing AR application may serve useful for other developers who wish to create a similar programming system.

Another contribution is the construction of the ofxWebViewInterface class, which provides a simple interface to build bidirectional communication with a Web View into any iOS app built with C++. Developers can use this interface to tightly integrate native iOS SDKs with web browser functionalities to build hybrid web applications. It could, for
example, be used by a future researcher who wishes to build an AR web browser with a different feature set.

While integrating web services with a data flow VPL was already possible with a system like Node-RED [62], my design is unique in its usability for non-developer audiences. I explored the accessibility tradeoffs between creating a general system that is easy for services to conform to, vs a restricted system that can provide more specific, easilyusable components for end users without domain knowledge to accomplish common tasks. In terms of prototypes these features enabled, the restricted, service-specific components proved to be more immediately useful.

On the topic of prototypes, some of my most significant contributions to the field of AR lay in pushing its practical boundaries with the prototypes I built using recently implemented features of the Reality Editor. While impressive AR prototypes have existed for decades, the fact that this diversity of experiences was able to be prototyped so quickly with the same tool is a testament to the Web platform. They serve as a launching point for other AR developers to build even more sophisticated and useful AR experiences, specifically by using Web to tightly integrate with existing systems and environments. Demos such as pulling live data from an API, or listening for broadcasts from nearby objects to show context-sensitive interfaces are simple to create with these tools.

I also explored the types of programs that can be build with the Logic Crafting system, which demonstrates the possibilities of allowing end users to customize their physical spaces with logical object interactions. Others can use this as a foundation to build more blocks, combine them into new types of programs, or even use the gaps in this system to inspire future programming systems that would better serve an IoT environment. For example, the lack of loops in the current system is a gap that needs to be filled in the future, else yield to a different system that supports periodic behavior not possible with Logic Crafting alone. I further explored the usability issues that arise from introducing a programming environment into an AR browsing application. Maintaining
consistency of the visuals and user interactions are nontrivial when the context of each view differs on such a fundamental level. The progress I made on this issue may serve as a starting point for future developers who wish to create coherent AR applications mixed with non-augmented screens.

A final contribution consists of novel explorations of permission-based security models for IoT objects in an AR context. Some findings include how to assign permissions, manage access to resources, and authenticate users with minimal intrusion to the AR user experience. As IoT and AR become more prevalent and coupled, this issue is sure to arise for more system designers.

7.6. Future Research

The system as implemented is far from complete. There are a number of small improvements that have been mentioned throughout the thesis where the system could be improved, such as improving the efficiency of syncing data across multiple clients. But more importantly, this thesis presents open questions and unexplored possibility spaces that present larger directions where future researchers can contribute. There are grand visions for this work that are not currently possible. But as technology improves, more objects become augmented, and the Reality Editor becomes more ubiquitous, these visions become more feasible and important to tackle.

7.6.1. Creating AR Interfaces

The AR interfaces displayed in the Reality Editor have all the capabilities of the web, but some of their capabilities are tied to their integration with the Reality Editor via the Hybrid Object API. The Three.js demo is possible because of the matrix subscription and fullscreen functions of the API, the context-aware demo relies on the visibility and global message broadcasting functions, and the menu-reactive demo relies on the interface state listener. As more AR interfaces are prototyped for more situations, this API can be extended to provide information from the Reality Editor that make the interfaces more powerful, or at least easier to create certain interactions. For example,
the code from the context-aware AR demo could be abstracted into an API function called `subscribeToVisibleInterfaces` which could trigger a callback with an updated list of visible interfaces rather than make users implement that functionality each time. There are also features that could be added to the Reality Editor itself to make the API more useful. The menu-reactive AR demo shows the potential for interfaces to change in response to buttons being pressed within the Reality Editor, but those buttons must be manually displayed by switching to them in the settings menu. Future work should explore how to automatically detect the context of the user to display a relevant menu for the interfaces in their surroundings. For example, the retail menu would show in a grocery store, an office management menu would show at work, and a home automation menu would automatically show when the user enters their home. One way to enable this would be to set a default menu for each WiFi network that displays when the phone connects to that network.

Another area not mentioned in this thesis but with plenty of room for improvement is a remaining bottleneck in the AR interface creation pipeline. Object markers must be generated and attached to objects via the closed-source Vuforia SDK, which cannot be automated for usability improvements. When open source technologies such as ARToolkit [63] become more mature, future work would involve switching to that system and creating an improved pipeline for generating objects from marker images. This would make the grocery demos work at an industry scale, where augmented products could automatically stay up-to-date with the store inventory rather than being individually, manually created.

### 7.6.2. Building Smarter Networks of Objects

Due to the amount of time this thesis spent improving the collective intelligence of networks by implementing Logic Crafting, I was able to identify a lengthy list of potential improvements for this system.
Logic Crafting is missing a major functionality that one would expect from almost any programming language: the ability to add loops. I paper prototyped such looping interactions, including blocks that would serve as "for loops", but there are several engineering problems that need to be solved before these become feasible. Currently, there is no way to introduce a real bug in the system via the logic blocks and links that you choose. The worst a user can do is prevent data from flowing from one object to the next because the links are not connected to the correct blocks or ports, or because they don't understand the functionality of a block correctly, such as the Switch or Seesaw blocks. Loops are currently disabled by the server, which checks for their existence when creating a new link to ensure they cannot be introduced to the system. But if they were to exist, they could easily be misused to introduce infinite data loops, which could congest the entire network and prevent any objects from working. This risk needs to be addressed before allowing them, but the system stands to gain a lot. For example, a procedural interaction like "blink the light on and off 20 times with 2 seconds of delay in between each when the button is pressed" cannot be created with a single logic node. The most efficient way to implement it would be to create 20 logic nodes that perform the action once and connecting them in series. This quickly becomes unreasonable.

Connecting to web services is one of the most powerful components of the system, but there is much depth to it that has not been explored yet. The webPost block has room for improvement. It attempts to enable Smarter Objects to connect with arbitrary web services, but uses a fixed format when sending data practically no web service would support unless intentionally built for or adapted to support the Reality Editor. A method for specifying the desired data format is required to make this more applicable. This could be implemented with a clever user interface, or by requiring users to type out the data format, or even by trying to detect the format the receiving API expects and letting users adjust it where needed by specifying other constant values. Determining a suitable method requires more research and iterative designs.

The system also lacks a few common methods of communicating with web services. Currently, it can create a webhook with the webListen block, but this is best suited to
process infrequent requests; it has not been tested with streams of requests, which would be better suited by opening a websocket [64]. Logic Crafting might benefit from blocks to send and receive data over websockets. Another missing block is one that makes an HTTP GET request when triggered. This could be called webGet, and could be used to fetch a value from an API on demand; for example, querying a weather API for the current temperature. These blocks would require unique user interfaces that can be prototyped in the future. There is also plenty of room to implement service-specific blocks analogous to TriggerIFTTT, but for other services. Even for services that can be reached via IFTTT, if may be beneficial to include specific blocks for them, as it removes the middleman and simplifies the process of setting up the connection. Building clean interfaces for useful, authenticated services would help users accomplish more with their objects.

7.6.3. Collaborative Editing Environment

A desired multiuser feature that is yet to be implemented, as described in Chapter 8.1.3, is the ability to save snapshots of entire system configurations that can be swapped between by multiple users of the system. This is even useful with a single user, as it would allow them to create a backup of the network configuration before experimenting and making changes. Having a version control system in the Reality Editor, even a primitive one, would encourage users to be more playful and experimental because it removes the risk of making mistakes. This feature becomes even more important when there are multiple users sharing the same space, as they can customize the network to suit their preferences without fear of causing problems for other users, who can restore their version of the network when desired. Taken to the extreme, this could lead to a branching version control repository for the network that can be manipulated similar to Git or Subversion to allow for safe, controlled, collaborative editing of a shared resource [65].

Another future improvement that this work enables would be the ability to save copies of logic nodes within a user’s Reality Editor device. Saved logic nodes could be used to
create multiple copies within the same network, to transport and drop them into another network, or to share and remix them. It is not hard to imagine having an online repository of shared logic nodes that users can download and add to their networks.

**7.6.4. Data Privacy and Security**

While my research into data privacy methods resulted in the current locking mechanism, much of my research remains unimplemented due to conflicting desires to maintain the simplicity of the interface while providing more powerful features. Future researchers might wish to investigate providing more flexibility in the user permission model. There are a number of actions that are all grouped into the same permission model, such as adding, linking, deleting, and viewing the contents of nodes and blocks. Ideally these could be specified separately, so that a user could, for example, allow others to link to their logic node but not let them change its inner program. Future work involves determining how to choose, assign, remove, and visualize these permissions without creating a bloated user interface and an overly complex conceptual model.

Future work could also try to fuse the fingerprint model with the passcode model to gain as many of the benefits of each while minimizing their limitations. It would be helpful, for example, to be able to authenticate with a fingerprint but belong to different user groups with associated permissions over different sets of resources. The fingerprint method also requires a method to restore authentication on a different device, which could perhaps be provided with a backup passcode. Even more important is improving the visualization of locked resources, which have a lot of associated data but only a single bit of information is displayed.

**7.7. Final Remarks**

The fact that all of these diverse experiences can be created, viewed, and interacted with the same platform is a testament to the potential of the Reality Editor. Other AR and IoT can support a subset of the demos in isolation, but we have created a general purpose bidirectional AR platform that fits into so many areas of a user’s life. From
grocery shopping, to home automation, to customizing industrial machines, to playing with toys, the Reality Editor adds utility when desired and remains hidden when appropriate. This diversity of use cases is the most impressive aspect of the Reality Editor. The preexisting platform, as described in “Smarter Objects: Programming Physical Objects with AR Technology,” was designed with a specific purpose in mind, namely “programming physical objects with AR technology.” But the Reality Editor has evolved beyond that. It now has the functionality for browsing and interacting with the physical and virtual objects found in all areas of people’s lives. It isn't about any one use case. It is a platform for browsing, customizing, and interacting with a seamless physical-virtual reality, regardless of when and where.

The Reality Editor is not a finished tool. The features I implemented improve its functionality, but the infrastructure for deploying this technology into the world as ubiquitously as the World Wide Web is not in place, and poses to be a significant challenge. However, the use cases demonstrated in this thesis justify such infrastructure, and the rapid progresses in image recognition and tracking, distributed sensors, and mobile AR browsers are bringing a world of bidirectional augmentation closer and closer to reality.
8. References


(40) "First URL active once more | Restoring the first website": first-website.web.cern.ch. 24 May 2017. <first-website.web.cern.ch/blog/first-url-active-once-more>.


(53) Enderton, Herbert, "Complete set of logical connectives" A mathematical
introduction to logic (2nd ed.), Boston, MA: Academic
(59) Johnson, Christopher W. "What are emergent properties and how do they affect the
(60) Juels, Ari. "Targeted advertising... and privacy too." Cryptographers' Track at the
(61) Birchall, Clare. "Radical Transparency?." Cultural Studies? Critical Methodologies
(62) Blackstock, Michael, and Rodger Lea. "Toward a distributed data flow platform for
the web of things (distributed node-red)." Proceedings of the 5th International
(63) Kato, Hirokazu. "ARToolKit: library for Vision-Based augmented reality." IEICE,
(64) Pimentel, Victoria, and Bradford G. Nickerson. "Communicating and displaying
(65) Ruparelia, Nayan B. "The history of version control." ACM SIGSOFT Software