Low Carb: The Virtual Breadboard Game

Aaron Gordon

California State University Monterey Bay

CST 499: Directed Capstone

Dr. Eric Tao

June 13, 2018

Executive Summary

This document is a report on the results of the Low Carb project, a proposed pedagogical tool for teaching computer architecture to those outside the computer science and electrical engineering disciplines. Despite the modern ubiquity of microcomputers their operation remains a mystery to most. The project sought to develop a framework consisting of a rudimentary hardware simulator and associated game-like user interface that could serve as the foundation for the creation of a puzzle game intended to walk players through the creation of a simple arithmetic logic unit. By creating such a piece of machinery, themselves, albeit in a simplified, virtual environment, it is hoped that players will gain insight into how the computers they interact with every day function. Although such a puzzle game is yet to be created the framework developed during this initial stage of the project shows promise both from a technical and entertainment standpoint. Its application as a pedagogical tool remains to be determined.

Table of Contents

Introduction

The Low Carb project is a unique pedagogical approach to teaching the subject of computer architecture: a rudimentary hardware simulator presented to end users via a graphical user interface as a puzzle game. Users are introduced to basic circuit design and logic gates through a series of puzzles that task them with building elementary computer architecture components using only previously built components. Once a complete set of logic gates has been built this way players have everything needed to construct an arithmetic logic unit - the "brain" at the heart of any modern computer.

The hardware simulator and associated user interface and puzzles are intended for the education of beginning computer science students – young and old – as well as members of the public at large about the topic of computer architecture, defined for our purposes as the arrangement of elementary logic gates into a functional computer. Users who lack the technical expertise requisite to the understanding of academic texts on computer architecture, those who lack access to the physical components necessary to experiment with physical breadboards and microchips – whether due to financial, geographic or other circumstances – and those who find current educational materials on the topic otherwise unsatisfactory are all positioned to benefit from the described application.

Issue in Technology

Microcomputers have become an inescapable and indispensable component of modern living. Our daily encounters with microcomputer architecture are now so frequent they are easily overlooked and taken for granted. Despite this, or perhaps precisely because of it, a great ignorance of the structure and operation at a mechanical level of the typical microcomputer

persists in the general population. Although an understanding of a microcomputer's architecture is not necessary for its successful operation the ignorance thereof is a glaring omission in the general education of today's well-rounded student.

Startlingly, Kehagias (2016) reports that of four categories of computer architecture undergraduate course assignments, the category pertaining to the construction of microchip logic gates was the least represented, appearing in only 13 of the 40 courses surveyed. This suggests even many beginning computer science students, and not only the general populace, are underexposed to the topic as well.

The way the human heart pumps blood through the circulatory system or the heliocentric nature of our solar system are processes understood by those outside the disciplines of medicine and astronomy, yet the manipulation of digital data via logic gates fundamental to modern computing is poorly understood even by many within the computer sciences. Perhaps this reality is a result of the relative youth of computer science as a discipline, or perhaps it's because our tablets and our phones interest us far less than our bodies and our place in the universe. Regardless for the reason, it is clear from both anecdotal and empirical observation that among the sciences, computer science in particular is poorly understood.

This problem has not gone unnoticed. A wealth of textbooks, popular science books, do-it-yourself microcontroller kits, children's toys and even computer games have been produced throughout the years with the goal of educating the public about computer science. Each has its strengths and weaknesses, and each is limited by the nature of its medium.

Textbooks are excellent sources of learning, but typically require a wealth of prerequisite knowledge that makes them inaccessible to the layman or the beginner. Popular science books are accessible in a way text books may not be but cannot provide the hands-on experience of an

academic laboratory environment. Microcontroller kits and board games are much more tactile than books yet tend to simplify or obfuscate important concepts for the purposes of ease-of-use; microcontroller kits can also require prohibitively expensive investment in physical components. Computer games can provide a limitless supply of virtual components to play with, but play is precisely what most prioritize at the expense of educational value.

Perhaps most conspicuously, presently available computer science education resources intended for laymen are focused on a single element of a computer's operation, such as electrical circuits or writing microcontroller code. It would seem only resources targeted at the technician seek to describe the computer's physical operation in its totality.

For the layman, and even perhaps for many beginning computer scientists, the topic of microcomputer architecture must surely seem like the inscrutable domain of tweed-clad academicians and Silicon Valley genius-entrepreneurs. In truth, of course, the fundamental building blocks of a computer – logic gates – are as simple in their operation as they are elegant in their application. Our project seeks to communicate this truth to its users by presenting the fundamental concepts of microcomputer architecture in an intuitive style and apart from the distracting trappings of a broader software engineering context, with the technical accuracy of an academic textbook, the accessibility of toys and games and the tactile feel of a physical breadboard.

Project Goals

A set of four specific goals was identified at the outset of the project to steer its design and development towards achieving the overarching purpose of educating laymen and beginning computer science students in the basics of computer architecture. These goals were: balance

technical accuracy with accessibility, provide an intuitive machine-human interface, prioritize education over play, and minimize upfront investment.

Balancing technical accuracy with accessibility meant requiring no prerequisite knowledge from our users and requiring no access to out-of-application resources, such as guides or manuals, without sacrificing the desired precision of the hardware representation presented to users.

Providing an intuitive machine-human interface meant forgoing the use of a hardware definition language or similar programming-oriented paradigm for arranging logic gates. Typical hardware simulators are operated by writing code in a language specific to that simulator. While our hardware simulator may be programmed directly via its implementation language of C#, satisfying this goal required implementing access to the simulator's full feature set through a graphical user interface.

Prioritizing education over play meant designing the application as an instructional tool and not as a computer game, despite the presence of elements borrowed from computer games in the design. Peripheral elements typically found in games, such as player resources and contextualizing narratives, were purposefully left out.

Lastly, minimizing upfront investment meant implementing the final product to give users immediate access with as little commitment of finances or setup time as possible. This meant both distributing the final application as freely as possible and packaging it in a quickly executable format and with minimal runtime dependencies.

Project Objectives

Satisfying the above goals was accomplished by fulfilling three primary implementation objectives, the details of which we now discuss. The first objective was the implementation of a rudimentary hardware simulator as a C# library; the second the development of an ergonomic graphical user interface for controlling the hardware simulator; the third the design of a series of puzzles that Socratically lead the player through the construction and use of elementary microcomputer components.

To satisfy the accessibility requirement of our first goal, this first iteration of the hardware simulator is not clocked (i.e. does not support flip-flop gates). Conversely, to satisfy the technical accuracy requirement of our first goal, the hardware simulator does support the construction and composition of logic gates, represented in their familiar microchip form. These microchips expose pins that can be connected to one another to transmit the electrical output from one chip to the input pin of another. A selection of built-in chips has been provided with the simulator to accommodate the construction of more advanced gates, and each of these chips behaves in accordance with its real-life counterpart.

Our second objective, developing an ergonomic graphical user interface, was laid down explicitly to satisfy our second goal of providing an intuitive machine-human interface. As previously stated, the typical hardware simulator is run by providing coded inputs, written in a hardware definition language, that logically describe the structure of a hardware circuit. This is clearly less than intuitive for the layman or the beginner, so instead a drag-and-drop user interface was provided.

The final two goals outlined above were satisfied through our third objective: the design of a series of puzzles that lead players through the creation of basic computer architecture

components. To achieve our third goal and prioritize education over play it was decided to present these puzzles in their literal form, explicitly as puzzles, rather than attempt to hide them behind a fictional setting or narrative. Meeting our fourth goal of minimizing upfront user investment also means our puzzles, whenever possible, stand-in for textual explanations. To give an example: rather than lecture players on electric flow and input-output paradigms, the first puzzle is solved simply by connecting a static output pin to another static input pin. Essentially, teaching the user interface has itself been reduced to a series of puzzles.

Impacted Community

This project is targeted at two groups: the non-technical layman interested in understanding his or her computer's operation on a physical level and the beginning computer science student interested in understanding the hardware-software interface (i.e. how does electricity become executable code). Both types of individual are assumed to be familiar with basic computer operation but unexposed to technical concepts such as circuit design, boolean logic, boolean arithmetic, hardware description languages, machine languages, etc. A third category of user - recreational players of puzzle games - may also show interest in the project for its value as a series of brain teasers, however this project was not pursued with them explicitly in mind.

The final executable delivered by the project is intended to provide a framework upon which an accessible, holistic instruction in the physical construction of a digital computer can be offered. As discussed above the ubiquity of digital computers is unbalanced with popular understanding of their operation. Though many of the materials discussed in the literature

review to follow already provide similar instruction, none appear to offer technical accuracy, accessibility and a tactile, hands-on experience in a single package.

Feasibility

Neither outreach to the public nor the application of the computer itself are novel approaches to computer science education. We have identified a collection of excellent, presently available and proposed resources for computer architecture education that share one or more of the stated goals of this project. These resources are discussed below in association with the strengths and limitations implicit in their respective mediums. We conclude the discussion with a brief justification of our proposed approach.

Literature Review

Perhaps the most obvious resource one turns to in the pursuit of knowledge is the book. An important distinction should be made between textbooks written for academic purposes and what we have labeled popular science books, intended for public, non-academic consumption.

One of the more popular textbooks on computer architecture is Patterson and Hennessy's Computer Organization and Design (Patterson, 2018). This is a thorough introduction to the hardware-software interface that touches not only on the physical properties of the microchips and microprocessors that power digital computing but on their manufacture and fabrication, as well. Alas, Patterson and Hennessy's text, written as part of the duos teaching at Berkley, makes a great number of assumptions about the knowledge of its readers – knowledge which has not been assumed in our users.

A less presuming text is Nisan and Schocken's The Elements of Computing Systems (Nisan, 2005). This text assumes little about its readers preexisting knowledge of Boolean logic and arithmetic but emphasizes hands-on labs that require programming in hardware definition languages. The lab work is integral to the book's pedagogical approach, but would no doubt prove too intimidating to the typical layman who's never written a line of code.

List of Reviewed Educational Resources	
Title	*Medium*
Arduino	Microcontroller
Autodesk Circuits	Software
Ben Heck's Circuit Board Game	Game
Circuit Scramble	Game
Code: The Hidden Language of Computer Hardware and Software	Text
Computer Organization and Design	Text
Elements of Computing Systems, The	Text
Minecraft	Game
Raspberry Pi	Microcontroller
Shenzhen I/O	Game

Table 1: List of Reviewd Educational Resources

A plethora of popular science books on computer architecture exist. Popular and typically well-reviewed on the Amazon.com bookstore is Code: The Hidden Language of Computer Hardware and Software by Petzold (Petzold, 2015). This text is targeted specifically at the public and draws on well-known historic analogies like Morse code to explain its concepts. However, Petzold's work is limited by its medium: as powerful as the written word can be it often fails to communicate what a more tactile experience is able to.

To that end several mini computers and microcontrollers intended for educational purposes have been developed. The two most popular representatives of each category are the Raspberry Pi and the Arduino, respectively. Both products offer users the opportunity for hands-on experience with digital computing at a much lower level than the typical consumer product. Yet this experience is not quite low level enough – both products effectively function as black boxes with programmable inputs and outputs that, while powerful, fail to expose the underlying physics at play. In addition, by virtue of being physical products there is a steep, often prohibitive cost to experimenting too brazenly with either, as any broken parts need replacing. This can prove especially prohibitive in a K-12 academic environment where children are as reckless as funding is scarce.

Attempting to reduce the cost of experimenting with physical hardware several developers have begun creating virtual breadboards. Applications such as Autodesk Circuits allow users to drag and drop electrical components into a virtual workspace and then simulate their behavior. These tools provide a low cost (often free) means of learning to work with microcontrollers like the Arduino and do an excellent job of digitally replicating the tactile experience of wiring input and output pins. The same limitations of microcontroller boards apply to these programs as well, though: namely obfuscation of the hardware-software interface for ease-of-use.

Countless toys and games have been produced that attempt to further distill the hands-on experience of boards like the Raspberry Pi and Arduino into something simultaneously fun and educational. Popular with young children are “circuit building blocks” that allow the fabrication of various circuit designs by snapping large metal connectors of predetermined length to one-another. Particularly noteworthy is a board game designed by Ben Heck that focuses on logic

gate design, though unfortunately the future production of said game remains uncertain (E., 2014).

Unsurprisingly, computer game programmers have at times also taken inspiration from the discipline of computer architecture. The popular building game Minecraft by developer Mojang includes a special kind of construction material called "Redstone" that behaves like a semiconductor (Elliott, 2017). Puzzle games such as Suborbital Games' Circuit Scramble present players with prebuilt logic gates that must have their inputs properly arranged in as few moves as possible to achieve a desired output (Staalduinen, 2016). Perhaps most like this project are the games of developer Zachtronics. In Shenzhen I/O players assume the role of a computer engineer tasked with designing and programming integrated circuits.

All these games, though – both cardboard and digital – are preoccupied with play. Minecraft's Redstone, for example, is just one of many kinds of bricks players can place in their world, making its usefulness as an educational tool limited to a carefully curated play session. Zachtronics' games emphasize the experience of working as a programmer in the early days of the industry; indeed, many of the games are intentionally obtuse precisely to achieve an atmosphere of confusion. Shenzhen I/O includes a 30-page manual of datasheets and assembly language specifications players are encouraged to print out for the authentic 1980s programmer experience. The common thread, then, is that these games require a teacher to turn them into educational experiences.

Justification

The computer games described in the literature review above have been used to construct impressive logic gates, including a functioning word processor (Savage, 2015). However, these

constructions typically take months, sometimes even years, and when finished are less than performant. To avoid these usability pitfalls and guarantee technical accuracy to real world circuits we have implemented and rigorously tested a legitimate, though rudimentary, hardware simulator as a standalone software library and wrapped a user interface around it, rather than attempt to implement microchip logic directly in the user interface as a computer game traditionally might.

The decision to exclude a hardware clock from our simulator may seem at odds with our desire to achieve technical accuracy. There can be no argument that the hardware clock is a crucial component of a microcomputer. Indeed, without it the architecture cannot support basic registers or memory. However, a clock is not required to construct an arithmetic logic unit and its omission greatly simplifies the initial presentation of logic gates. To wit, most computer architecture textbooks do not introduce the hardware clock until later chapters. Although absent from this first iteration, a hardware clock and flip-flop gates are likely candidates for implementation in future versions of the simulator.

The textbooks described in the literature review above teach the construction of logic gates via the industry-standard method of writing code in a hardware definition language. We broke with this approach and opted instead for an intuitive interface built around the drag-and-drop gesture. The drag-and-drop gesture is both familiar to users and readily portable across interface devices (e.g. mice, touch screens) making it an ideal mechanism for user interaction with our hardware simulator. Elementary chips may be dragged from a side panel and dropped into position on a virtual breadboard. Output pins may be dragged to input pins and dropped to wire two chips together. These wires are then able to display the signal (low or high) traveling

through them as the simulation runs – another benefit of our graphical user interface over a hardware definition language-driven program.

To make the drag-and-drop approach to constructing logic gates as intuitive as possible, and to encourage experimentation and minimize user frustration, it was important to allow modification of logic gates while the simulation is running. To that end our hardware simulator takes a somewhat unorthodox approach of modeling the state of every wire and pin in the gate at every time step. This sort of a naïve algorithm is both unnecessary and potentially performance-limiting in a commercial hardware simulator. That said, our simulators purpose as a tool for constructing elementary chips relieved us of most performance-related concerns, and this approach perfectly satisfied our user interface requirements.

Perhaps where our project has diverged the most from those discussed in the literature review is in the use of puzzles as our primary instruction mechanism. We believe structuring our lessons as puzzles helps with approachability and retention – users presented with a puzzle implicitly understand what they're attempting to learn is difficult but solvable. Puzzles also have a way of holding human attention, demanding a solution to their problem. Finally, we firmly support a Socratic, inquiry-based approach to education that encourages learners to find their own answers by asking the right questions. Puzzles are a natural fit to this paradigm.

Design Requirements

We turn now to the discussion of the design requirements initially specified for our project. All requirements fell into one of four broader categories: hardware simulation, chip validation, toolchain support and user interface design. We first look more closely at each of these categories then review the criterion selected to measure proposed functionality in each

category against. We then enumerate the final deliverables comprised by the proposed functionality and consider the methodology of their implementation. We conclude this section with a reflection on ethical and legal considerations of the project.

Functional Decomposition

The functional requirements for the hardware simulation category are listed in **Error! Reference source not found.**. This category consists of those requirements we deemed constituted the minimum functionality necessary for a viable virtual breadboard. Naturally, an interface for placing chips and connecting their pins was a requirement, as was a mechanism for solving the resulting circuit. Less typical was the requirement to support iterative solution of a circuit, necessary for us to support simulation-time interaction with the circuit through the user interface. Finally, to make the hardware simulator useful "out of the box" it required the inclusion of common, "built-in" chips.

Hardware Simulation Requirements
1. Placement of chips on a board.
2. Connection of pins between chips.
3. Provision of common chips as built-ins.
4. Solving a circuit simulation.
5. Solving a circuit simulation iteratively.

Table 2: Hardware Simulation Requirements

Both the built-in chips provided with the simulator and any chips developed by users of the simulator required a mechanism for testing their accurate operation. To this end we specified the following requirements, listed in **Error! Reference source not found.**, as part of the Chip

Validation category: reading, iterating through and checking the actual against the expected output of a contract file from disk. A contract file is simply a list of input signal combinations and expected output signal combinations for a chip. Chip validation requires testing each listed input combination on the actual (or in our case, virtual) chip and checking that the actual output from the chip matches the output listed in the contract.

Chip Validation Requirements
1. Read a contract file from disk.
2. Iterate through a contract file's listed inputs.
3. Validate actual outputs against a contract file's listed outputs.

Table 3: Chip Validation Requirements

Although simple in premise, a chip contract file can grow very large for chips with multiple input pins, since every input combination needs to be tested. Writing each of these contract files by hand would have been a tedious and time-consuming enterprise. As such we identified a category of feature requirements specific to easing our own development work. This category of functional requirements, which we've labeled as Toolchain requirements, are not experienced by the end user but were no less important to implement.

Toolchain Requirements
1. Automatically generate contract files from a chip's source code.
2. Automatically generate unit tests for using a contract file.
3. Automatically include contract files and unit tests in source code collection.

Table 4: Toolchain Requirements

Toolchain requirements are listed in **Error! Reference source not found.**. All three functional requirements meet the goal of speeding up the development process by minimizing tedious and redundant work. The requirements taken together describe a pipeline of processes automatically run after the source code for a chip is written. The output of this pipeline is a contract file and a unit test for the chip, both of which are imported into the management tools for the project's source code (in this case a Visual Studio solution).

Error! Reference source not found. lists the final category of requirements, pertaining to the functionality of the user interface. The key takeaway from these requirements is that the user interface is operated primarily through drag-and-drop gestures and should be synchronized at all times with the underlying hardware simulator, even if the circuit is modified by the user mid-simulation.

User Interface Requirements
1. Placement, connection of chips via drag-and-drop gesture.
2. Display pin, wire values as calculated by simulator.
3. Support modifying circuits mid-simulation.

Table 5: User Interface Requirements

Thus far we have described only the necessary behavior of our functional requirements. In the subsequent section we examine those factors which restricted how we were able to satisfy these requirements.

Design Criteria

The following six criteria were identified as dimensions along which the implementation of each functional requirement should be measured: performance, cost, distribution, human interface, code complexity and reliability.

We were fortunate as regards the criterion of performance insomuch as our hardware simulator is simulating far less complex circuits than one intended for industry-use. That said, we were not fully free from concerns over application performance, as one of our goals was to limit upfront investment, which includes investment in high-end computer hardware. Given that one of our target audiences is the public education sector, where funding for new computers isn't always available, it was important that the final project perform well on even modest or older hardware.

On the topic of cost, our own budget was effectively non-existent. Moreover, as our intent has always been to distribute the final application freely, charging end users to recoup any financial investment in development wasn't an option. The financial cost of any potential implementation, then, was a serious factor of consideration throughout development.

Of course, users will refuse to use even a free application if setting it up is a lengthy or otherwise unwieldy process. The ease of distribution was always a concern when evaluating technologies and tools prior to and during development. Dependencies on third-party libraries were kept to a minimum, and where dependencies were unavoidable we always tried to rely on common dependencies users were likely to already have. Another concern with distribution was precisely how users would procure the application download and how long that download would take.

Although setup is a user's first experience with an application it's still important the application itself be user-friendly. Human interface was an important restricting criterion on the implementation of functional requirements. The interface needed to be tactile – interacting with a virtual breadboard needed to feel as like using a real breadboard as possible. The visual presentation of simulation elements was an important consideration as well. Graphical elements needed to be clear enough to communicate their function, simple enough to run on integrated GPUs (or even software fallbacks) yet flashy enough to hold the interest of users as they struggled through solving a particularly challenging puzzle. Most of all, the user interface could not rely on hardware definition languages, programmable chips, complex menus or any other common element of industry simulators.

The code powering the application is not directly apparent to the end user, yet the complexity of the codebase was given equal consideration as the complexity of the application itself. The primary limiting factor relating to the code complexity criterion was the short schedule for the project. To avoid losing too much time to complex implementation issues it was necessary to judge each architectural decision by how much complexity it added to the codebase.

The final criterion influencing implementation of our functional requirements was reliability. Reliability of the codebase was important for every facet of the project, but particularly crucial for the hardware simulator itself. Before any code could be added to the project a test plan for validating it in the form of a unit test was first required. Moreover, when choosing tools and frameworks to work with, mature, well-documented and provably stable ones were given priority over new or untested ones, no matter how exciting or cutting-edge they may have been.

Each of these criteria helped to steer decision making as functional requirements were implemented. The final success of the project depended a great deal on adhering to the limits set by each of these criteria whenever a decision about implementation details needed to be made. This was especially true with the architecture of the hardware simulator, where complex implementations capable of far more advanced simulation than was necessary for our project were a constant temptation.

Final Deliverables

Four final deliverables were generated at the conclusion of this project: a hardware simulator software library, a command line interface application for extending the development toolchain, an interactive user-interface for operating the simulator and a single website for describing and distributing all three.

The hardware simulator software library took the final form of a Microsoft Dynamic Link Library (.DLL) file. The library was implemented in C# and is intended for use with other C# and .NET / Mono applications. The “virtual board” and its constituent components are exposed to consumers of the library via public interfaces in the form of C# classes. This library has been freely open-sourced via GitHub.

The command line interface application is a simple executable that implements the Toolchain category of functional requirements. The tool itself is development environment-agnostic, though we have chosen to integrate it with our chosen development environment of Visual Studio via a batch script included with the source code, both open-sourced on GitHub.

The playable puzzle game is a standalone executable generated by Unity 3D and dependent upon the hardware simulator software library. Unity 3D applications are compiled

against the Mono framework, which enabled development of the puzzle game in the C# language to expedite consumption of the hardware simulator software library. Like the software library, the code used to build the final executable has been freely open-sourced on GitHub.

Tying each of these deliverables together is a website used for distributing the playable game. The source code of the entire project has been made available to the public via open sourcing, however using this code requires end users to manually compile it. Given the target audience of the project is the non-technical layman this is an unacceptable distribution method. Instead a website has been created that provides download links directly to the standalone executable with a description of the project and its intended use.

Methodology

The successful execution of this project hinged on a thorough understanding of the technical material, careful consideration of a pedagogical approach and technical ability to execute on the development of the specific software artifacts.

The material discussed in the above literature review provided an excellent source for reviewing the topics of computer architecture encompassed by the project. Reading or rereading literature on the topic helped to clear up misunderstandings of the material and filled gaps in developer knowledge. More importantly, taking time to play with virtual breadboard software and computer games that draw their inspiration from computer architecture provided an opportunity to understand how other developers have tackled these topics, both from a technical implementation standpoint and from the important perspective of making the subject matter fun and engaging for players. This also provided the opportunity to identify where existing software

fails in educating its users, either by failing to cover specific elements of the proposed material or by overemphasizing game play or aesthetic over pedagogy.

Technical implementation began with research into the software architecture of industry hardware simulators. Fortunately, the hardware simulator required for this product was comparatively simple and so adequate literature was easy to come by. Craig's paper "Circuit Description and Elementary Hierarchical Circuit Simulation Using C++ and the Object-Oriented Programming Paradigm" (Craig, 1991) though antiquated by computer science standards provided invaluable insight into the foundations of hardware simulator architecture. Parallel to implementing a hardware simulator a graphical frontend to accommodate intuitive user interaction was developed.

This frontend was developed using Unity 3D, which was selected in part for its large, existing libraries designed to handle user interactions like mouse clicks and view scrolling. Unity 3D exposes its libraries for development in the C# language, which made the development of the hardware simulator as its own C# library an easy decision. Finally, Unity 3D ships with a powerful level editing tool that has assisted in the creation of puzzles and other gameplay elements.

Ethical Considerations

The primary ethical consideration during development was respect of the intellectual property of other programmers and educators. As evidenced by the extensive literature review performed above there existed prior a great wealth of educational material on the topic of computer architecture. Indeed, the very essence of this project was not the conveyance of new knowledge but of well-understood knowledge to a previously unreached group. As such there

were inevitable similarities between this project and existing texts and computer software. However, care was taken to minimize these similarities by the thorough application of a unique pedagogical approach to the topic, not only for the benefit of the user but out of respect to the originators of the material that has inspired this project.

It was also important to consider a broad group of individuals when developing the pedagogy of this project. The digital distribution of the final software means users may come from diverse socio-economic backgrounds with diverse, pre-existing knowledge of the topic. What may seem like common knowledge to an individual from one group can be a stunning revelation to that from another. It's important that presumptions of the user's knowledge are kept to a minimum.

It was beneficial to attempt to predict gaps in a hypothetical user's knowledge. The abstract idea that electric current "flows" like water through a pipe is not learned through everyday observation of electrons (were such possible) but in a grade-school classroom. Yet there are many who have not enjoyed the privilege of access to high quality grade-schools. The flow of current and other related ideas needed to be addressed clearly in puzzle design for these users.

No doubt other life experiences that help paint the mental models computer architects rely upon are often taken for granted. Though careful consideration was given to what these experiences might be and what lessons they might impart, and concessions for those who may not have had such experiences was made where possible, areas of oversight likely persist. Continuing to draw feedback from as broad a group of testers as possible may help in continuing to address this issue.

A final consideration, though minor, was distribution. Hoping the final application generated by this project may be fruitful for educational use it has been made available for download free of charge.

Legal Considerations

Three pieces of third-party software were leveraged in the development of this project: Adobe Photoshop, Unity 3D and Autodesk Maya.

Adobe Photoshop is an industry-standard image editing tool. Luckily the developers already hold a paid license for the commercial use of the application.

Unity 3D is freely available for commercial and non-commercial use with some limitations (notably the automatic inclusion of a Unity 3D splash screen at the startup of applications built with the platform). These limitations may be removed from the product via the purchase of a professional license. However, since these limitations posed no obstruction to the completion of our project the free version of the software sufficed.

An educational edition of Autodesk Maya is available to students for non-commercial use with no notable limitations. The proposed project meets the criteria of student work necessary for the use of the educational version, and the free distribution of the resulting software artifact insures the project remains non-commercial in nature.

Lastly, third-party software fonts were used in the development of the porject. Please refer to Appendix E for details on the licenses these fonts were used under as well as the licenses of the above software packages.

Timeline

One word best describes the timeline that was initially proposed for the project: ambitious. In retrospect the scope of the proposal was simply too large to be realistic and several issues that presented themselves throughout development resulted in planned features falling out of scope.

The original plan was to develop the simulation library in parallel with the Unity user interface application, creating broader and broader vertical slices of the project each week. The development of the hardware simulator proved more time consuming than expected, though, and little or no time was left at the end of each week for working on user interface features. Given the important of a user interface to testing and demoing the product other features were dropped one-by-one to try and leave enough time for developing in Unity.

First the implementation of low-level circuit simulation (i.e. transistor-based circuits) was pushed out of scope. Soon after it became evident that there wouldn't be enough time to both program the game and design the proposed suite of puzzles. Initially the set of puzzles was only reduced, then eventually puzzles were dropped from the plan completely and the project was restructured as the creation of a framework that a puzzle game could later be built upon.

Despite all these cuts, with only two full weeks of time remaining user interface work had barely even begun. Fortunately, by this point the hardware simulator was functionally complete and all efforts could be focused on developing the user interface. Despite this, there was legitimate concern leading up to the time of writing that the user interface could not be finished, and so work proceeded towards the creation of a "read-only" version of the interface that was fundamentally a mockup powered by the hardware simulator library. After a few very late nights, though, this mockup was turned into a first pass at a functional interface.

This description of events is detailed in the below table, which compares the original project schedule with notes adapted from weekly progress reports.

Week	Planned	Actual
1	Implement basic logic gates. Implement rendering of components, wires, voltages.	First tests showing proof of concept completed. Elementary built-in chips completed
2	Implement composite logic gates. Implement drag and drop behavior. Design advanced logic gate construction puzzles.	Hardware simulator architecture vastly improved. New design based on techniques used by commercial applications.
3	Implement circuits. Implement support for new components in GUI.	Testing framework for validating virtual chip implementations finished.
4	Implement transistors. Design basic logic gate construction puzzles.	File baker utility implemented and build chain automated to generate chip contracts and unit tests.
5	Implement advanced logic gates. Implement human-readable displays in GUI. Design bitwise encoding puzzles.	All functional requirements of the hardware simulator library completed. Major milestone reached.
6	Implement ALU. Design ALU construction puzzles. Begin test-group surveys.	Dynamic chip mesh generation, camera controls and other foundational user interface work completed.
7	Design bit manipulation puzzles. Fix bugs. Adjust puzzles from survey feedback.	Initial implementation of chip creation and movement, pin connections, completed. Usability testing now possible.
8	Fix bugs. Adjust puzzles from survey feedback. Distribution, if applicable.	Usability testing, planning for future work.

Table 6: Planned and Actual Project Timeline

It truly isn't surprising that so many features had to be cut from the project when rereading the initial project plan. That said, there were no doubt a number of crucial mistakes made during development that exacerbated the problem.

Much of the work done in the first week ended up being thrown out and restarted in the second due to poor architecture. This could have been avoided with additional research into the methods involved in implementing commercial hardware simulators before programming began.

The adage to measure twice and cut once still rings true, even when you're "cutting" something digital.

It was understood early in the project that a system for speeding up the validation of chips was important. Regrettably the system that was planned proved much more difficult to implement than it had on paper. Most of weeks three and four were spent implementing what should have been a one-week feature, at most. This is a textbook example of the Sunk Cost fallacy; despite knowing valuable hours were being burnt past schedule work persisted in the vain hope it would "all be worth it". Alas, it probably wasn't.

A final mistake was overdesigning the user interface too early, both aesthetically and functionally. Instead of trying to implement the final vision of the interface in the first pass, it would have been much better to get a rough approximation that allowed basic interaction with the simulator up and running in time for usability testing.

Usability Testing

Regrettably the scheduling issues described in the preceding section prevented the execution of planned usability testing. However, additional work on the user interface was completed in the days leading up to the time of writing and it is hoped that some modified form of usability testing can be performed and an addendum to this document describing the results published.

Final Implementation

At the highest level the entire Low Carb project has been divided according to a Model View Controller paradigm. Model View Controller, commonly referred to as MVC, is a design pattern that advocates for the separation of the internal structure of data (the model), the presentation of the data (the view) and operations upon the data (the controller) from one another. In our case, the model is encapsulated entirely within the Low Carb simulation library (along with functionality for validating the model) while the view and controller portions are implemented in the Unity application.

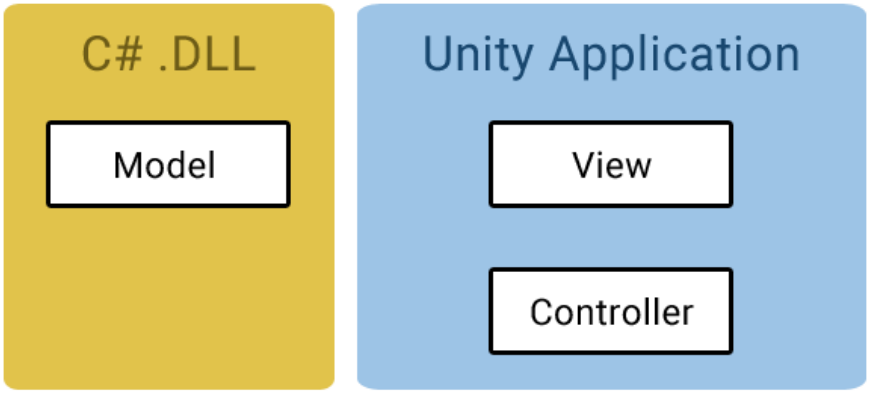

Figure 1: MVP Framework Applied to Low Carb Project

Model

Modeling a hardware circuit and its behavior is a primary requirement of the Low Carb project. It was decided early on that the hardware simulator should be encapsulated in its own library both to facilitate ease of development and to provide opportunities for future reuse of the logic.

Pins

The atomic component of the hardware model is the `Pin` class, which corresponds to a single port (input or output) on a chip. A pin contains only two data members, a string label and a signal (an enumerated value of low, high or unknown). Pins may be instantiated with a label which is made available to consumers of the model for UI display purposes either by directly passing a string or via an implementation of the `IlabelGenerator` interface. A label generator is useful when declaring multiple pins with similar names that follow some known pattern, such as “Input A”, “Input B”, and so on.

Chips

Pins are used to construct chips. A chip is declared as an implementation of the `IChip` interface, and as such must implement the `GetInputPins`, `GetOutputPins` and `Simulate` methods. `GetInputPins` and `GetOutputPins` each return an iterable list of pin object references (where the pin objects themselves typically exist as members of the implementation class). `Simulate` is called once per simulated time step and is responsible for updating the signal value of the pins referenced by `GetOutputPins` in response to the signal values held in the pins referenced by `GetInputPins`, reflecting whatever logic the chip is implementing, such as a logical “and” combination of the inputs. If `GetInputPins` and `GetOutputPins` describe the public interface of the chip, `Simulate` describes its behavior.

The first iteration of the simulation library includes a collection of 1-bit implementations of common logic chips. However, anywhere a chip is used throughout the library it is referenced

as an `IChip` instance. This allows us to seamlessly integrate new built-in chips in future versions and provides users of the library the opportunity to implement their own chips. This approach also allows a chip to be defined as a composition of other chips, a common practice in circuit design.

Board

Although a chip can simulate a data transformation on its input pin values on its own the results are typically unexciting. Where circuit design becomes interesting is in the combination of multiple chips. The Low Carb simulation library defines a `Board` class for organizing and connecting chips. Chips are added to a board object by passing the chip's type to a templated `AddChip` method which instantiates the appropriate chip and returns an integer handle to it. This was done to provide a layer of abstraction to consumers of the library that relieves them of the responsibility of managing references to chips. If the instantiated chip object is needed by a consumer, e.g. to check the label or signal value of its pins, the board class provides a `GetChip` method that accepts a chip handle and returns a reference to the chip object.

The board class also exposes the `ConnectPins` and `DisconnectPins` methods for managing the wiring between chips. These methods manage the internal wire collection of the board and accept instances of the `PinHandle` class, which is a plain old data structure that indexes a pin on a board via the combination of the pin's type (i.e. input or output) its owning chip's handle, and its index within the chip's list of pins.

Chip connections are represented by instances of the `Wire` class. Wire stores a pair of pin handles, source and destination, and performs all necessary validation logic when setting

these values, such as guaranteeing source and destination pins do not belong to the same chip. Wire instances associated with a board are stored in a `WireCollection` class that indexes its elements according to their destination pins. Since only one wire may feed into an input pin this approach guarantees a unique index for each wire. Wire objects are instantiated by the board class and added to its internal wire collection whenever a chip is added to the board and deleted when their corresponding chip is. Doing this insures that whenever a user requests a connection through `ConnectPins` a wire is already available to store the connection, and until such a request is made the wire's source is left empty, opting the wire out of any simulation events.

Simulation

Two approaches exist to simulating a board that has been appropriately configured with chips and wires, both accessible through methods on the board class itself. `StepSimulation` performs a complete iteration over each of the chips on the board for a single unit of time. `SolveSimulation` repeats this process until the circuit described by the board reaches an equilibrium where further simulation steps will not cause any change in state.

Model Validation

The accuracy of the overall simulation is only as trustworthy as any individual chip's Simulate method. It's thus crucial that all chips used on a board be thoroughly validated, including the collection of built-in chips provided with the library. However, the process of manually writing tests to validate chips is tedious and time consuming, and so a validation framework has been implemented as part of the simulation library. This validation framework is exposed via the library's public interface, allowing consumers of the library to leverage the framework to test their own chip implementations, as well.

Chip validation involves setting a chip's input pins to various combinations then checking that the actual values of the output pins match expected values. These input combinations and their corresponding expected output values are stored as "chip contract" files. A chip contract is a plain text file, which can be easily read and edited in any text editing application, that simply lists one pairing of input pin combination and expected output pin values, separated by a delimiting pipe character, on each line of the file. Representing test inputs and their expected values this way makes designing and iterating upon tests quick and easy.

The contract file is read at runtime using the `ChipContractFileReader` class. This class loads the contract file from disk and instantiates a `ChipContract` object to serve as an in-memory representation of the contract. These contracts can then be passed to a `ChipValidator` object alongside an implementation of the `IChip` interface, against which the contracts are validated. This validation framework so simplifies the process of testing chip implementations that it was used extensively in writing the unit tests for the simulator library.

File Baking

Although the validation framework relieved much of the tedium involved in unit testing chips a great deal of work was still involved in creating chip contract files and writing unit tests to load and validate the files. This work was automated via the creation of a command line tool responsible for automatically generated – or "baking" – contracts and their containing unit tests.

The file baker tool has a large body of boilerplate code responsible for parsing and verifying arguments passed to it on the command line that lacks the requisite novelty to merit its inclusion in this discussion. Suffice it to say that once these arguments have been parsed an implementation of an `ICommand` interface is selected and executed.

The `ContractsCommand` implementation accepts a C# source file as input. The source file is compiled at runtime using the reflection capabilities of the C# language and information regarding its input and output pin structure is loaded into memory. This information is then used to create a template chip contract file, consisting of lines populated with various input combinations and an appropriate delimiter, but no expected output values. This reduces the work involved in creating a contract file to the simple task of filling out expected values for a given set of inputs.

The `UnittestCommand` implementation generates a valid C# source file that can be used as a unit test for a given chip. The unit test expects the presence of both an implementation of the chip and a chip contract file corresponding to that chip, identified via a common naming scheme.

The `FillprojectCommand` implementation accepts a chip contract file or a unit test, i.e. the two types of file generated by the prior two `ICommand` implementations and inserts it into a preexisting Visual Studio project. This action can be performed while the project is open and being edited, essentially creating a method for "hotswapping" chip tests while in the process of implementing said chips.

Build Chain

The file baker command line tool exists as a standalone application that can be run by consumers of the simulation library as they see appropriate. For the purposes of developing the simulation library a batch script was written that automatically generated chip contract file templates and unit tests and added them to the simulation library's unit testing project after each successful build of the source code. This guaranteed that no built-in chips could be implemented

without an effective unit test suite and in turn helped to guarantee the overall quality of the simulator library. Moreover, this extension to the project's build chain vastly sped up the implementation of new chips allowing the project to ship with a healthy selection of built-in chips, further increasing the value provided by the product even in the tight eight-week schedule.

View

Just as the simulator library served as the high-level model of the Low Carb project a Unity 3D application served to encapsulate the view and controller portions. Unity 3D is a combination computer game engine, level design tool and scripting framework that's built in part using the .NET framework and thus is fully interoperable with C# dynamic link libraries, such as the simulation library previously described. This interoperability permitted the direct invocation of simulation methods from within code written on top of the Unity framework.

Unity is designed first and foremost as a framework for building games and other interactive applications. As such the framework relies heavily on an event system responsible for invoking various callbacks throughout the lifetime of an application and over the duration of a single frame. Almost all new code written for the Low Carb project was in the form of these callbacks, attached to preexisting Unity-triggered events.

Visualizing Chips

The Unity view renders chips as 3D meshes, each of which is constructed dynamically at instantiation time by the `ChipMeshBuilder` class, using a collection of six component meshes:

- Left Casing Cap

- Right Casing Cap
- Casing Midsection
- Left Half-Pin
- Right Half-Pin
- Full Pin

Every chip contains a single left and right cap, as well as two left half-pins and two right half-pins (a pair for each cap). The number of casing midsections and additional full pins is determined using the `GetInputPins` and `GetOutputPins` methods of the `IChip` implementation the chip mesh is being assembled for.

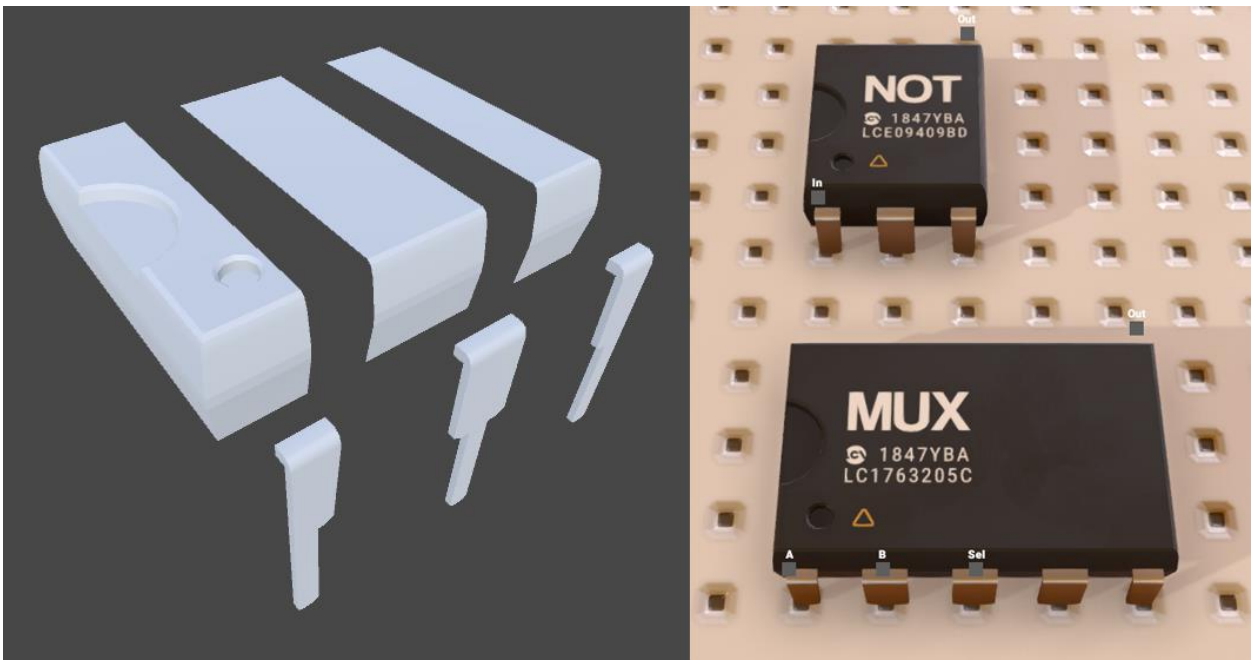

Figure 2: Individual Chip Meshes and Variable Length Constructed Chips

Although this approach allows the generation of chip meshes of variable lengths directly from a chip's public interface it introduces a wrinkle related to the materials eventually applied to the mesh. Specifically, the texture lookup coordinates stored with the vertices of the chip

casing midsection mesh are the same for each instance of the mesh. This results in the texture being applied to the mesh repeating over the body of the chip rather than spreading across it. To solve this the `ChipMeshBuilder` class calculates a texture coordinate offset for each segment using that segment's position within the overall chip to insure proper texture placement.

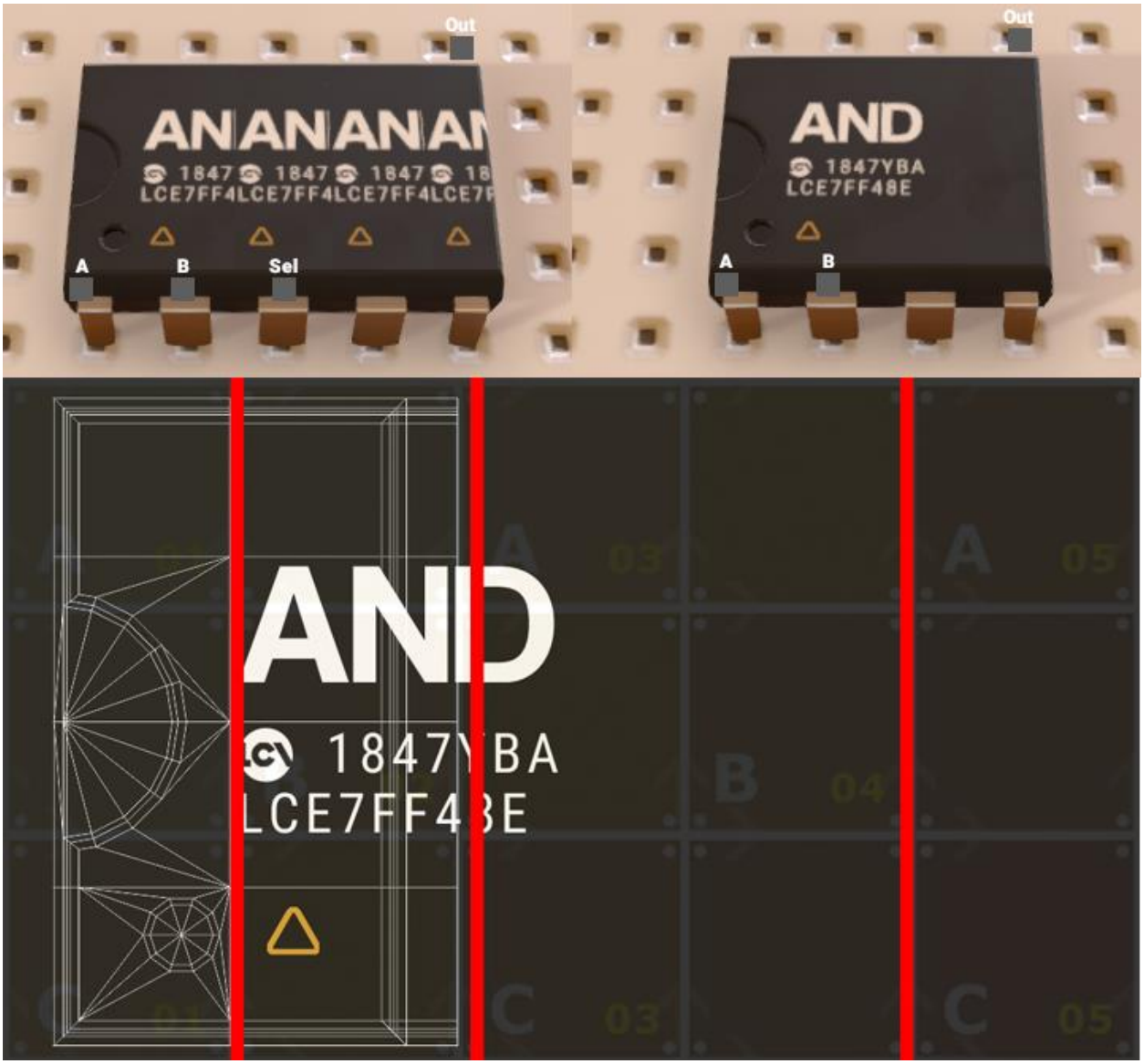

Figure 3: Chip Mesh Without and With UV Correction; UV Layout

The final noteworthy component of chip visualization is the drawing of pin icons. These icons are presently small colored squares "floating" above the pins they represent. Each icon is labeled using the label property and colored according to the signal property of the corresponding pin object within the simulator library. The color of the icon is updated each frame to reflect the most recent signal value calculated by the simulator library for that pin, providing immediate feedback to users as they interact with a circuit.

Visualizing the Breadboard

A piece of software subtitled, "The Virtual Breadboard Game," would be rather underwhelming with no *actual* breadboard! To this end a simple breadboard is included prominently in the game. To enhance the tactile feel of the game the breadboard's holes are perfectly aligned with the pin meshes to create the visual illusion of chips "plugging into" the board when placed. This was accomplished through the implementation of a `BoardPosition` class that represents a specific hole in the board as a 2D, integer position relative to the bottom-left corner of the board. Static helper methods for converting from a 3D world coordinate to a board coordinate and back were written to simplify the implementation of any class that interacts with the board.

Controller

Unity exposes multiple methods for displaying and attaching behavior to a user interface in an application. This allowed the creation of a controller layer directly on top of the view layer within the same Unity application.

Controlling Chips

Chips as entities, distinct from their pins, are relatively static objects within the simulation model, only ever being added to a board or removed from it. However, within the context of the user interface chips need an associated position to be drawn at, one that the user can modify for the purposes of "moving" chips around the board.

Chip creation is handled via a flyout menu that is drawn over the 3D game area. This menu lists the simulation library's built-in chips. Clicking a chip in this flyout invokes a method on the `GameChipFactory` class responsible for assembling the chip's mesh and pin icons as described in the previous section and instantiating a representation of the chip at the simulation library level for calculation purposes. Once assembled the chip mesh is drawn at the origin of the board, i.e. board coordinate (0, 0). Presently there is no method for removing a chip from the board, though this is planned as future work.

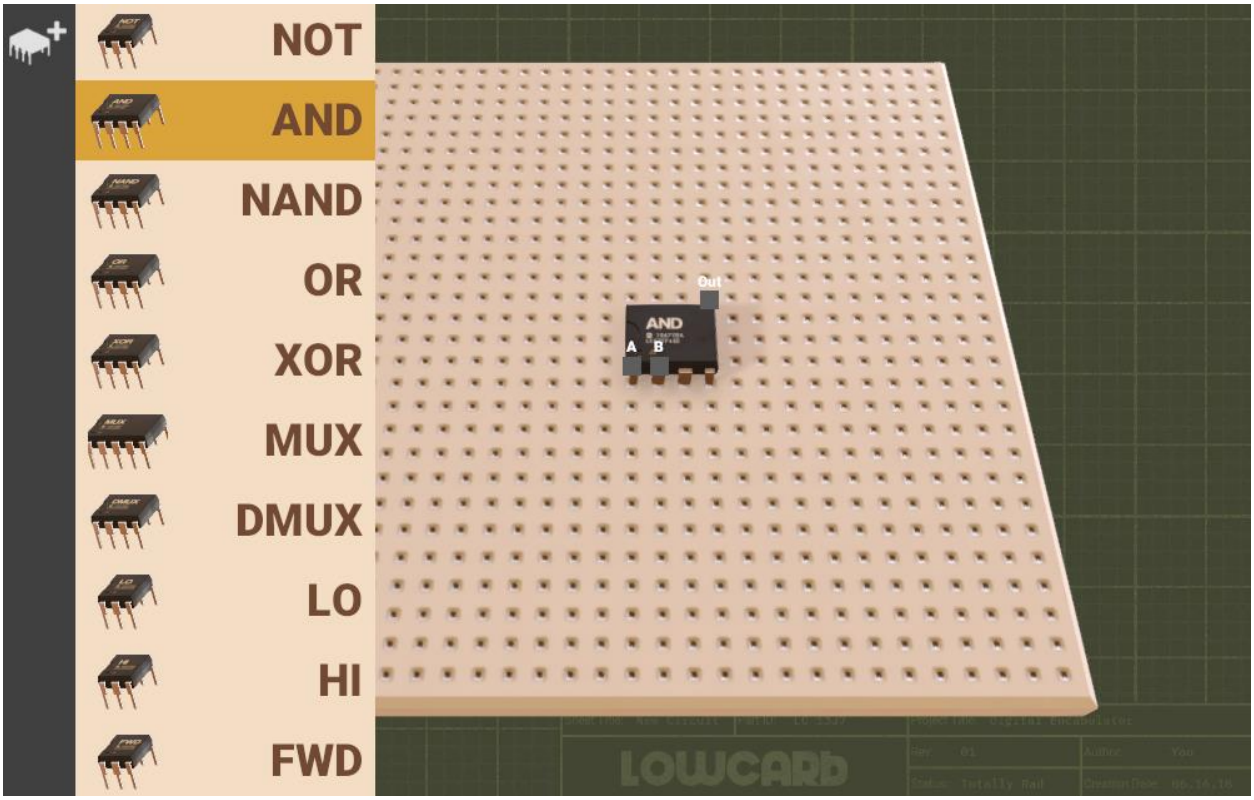

Figure 4: The Add Chip Flyout Menu with the "AND" Chip Selected

A created chip may be moved around the board by clicking on it once to “unseat” it then clicking anywhere on the board to reseat the chip at the new location. The chip’s new location is calculated by casting a ray through the mouse cursor and calculating where it intersects the board. This intersection point is then converted from 3D world coordinates to 2D board coordinates and passed to the chip. At time of writing chips do not check for collisions with one another, including when initially created at the board origin. This is an issue we’d like to resolve in future work.

Controlling Pins

The connection of output pins from one chip to the input pins of another is the crucial element that makes a hardware simulator functional. Unfortunately, due to time constraints and poor scheduling discussed elsewhere our implementation of this behavior is rudimentary (though functional) in its current state. The pin icons drawn above each chip may be clicked to establish them as a connection candidate. Clicking another pin icon attempts to connect it with the candidate pin. The logic for determining if a potential connection is valid is handled opaquely by the simulator, which throws a detailed exception if the connection is invalid. In the event the connection is valid the simulator connects its internal representations of the pins. Upon the next simulation step the pin icon colors are updated to reflect a low, high or unknown signal.

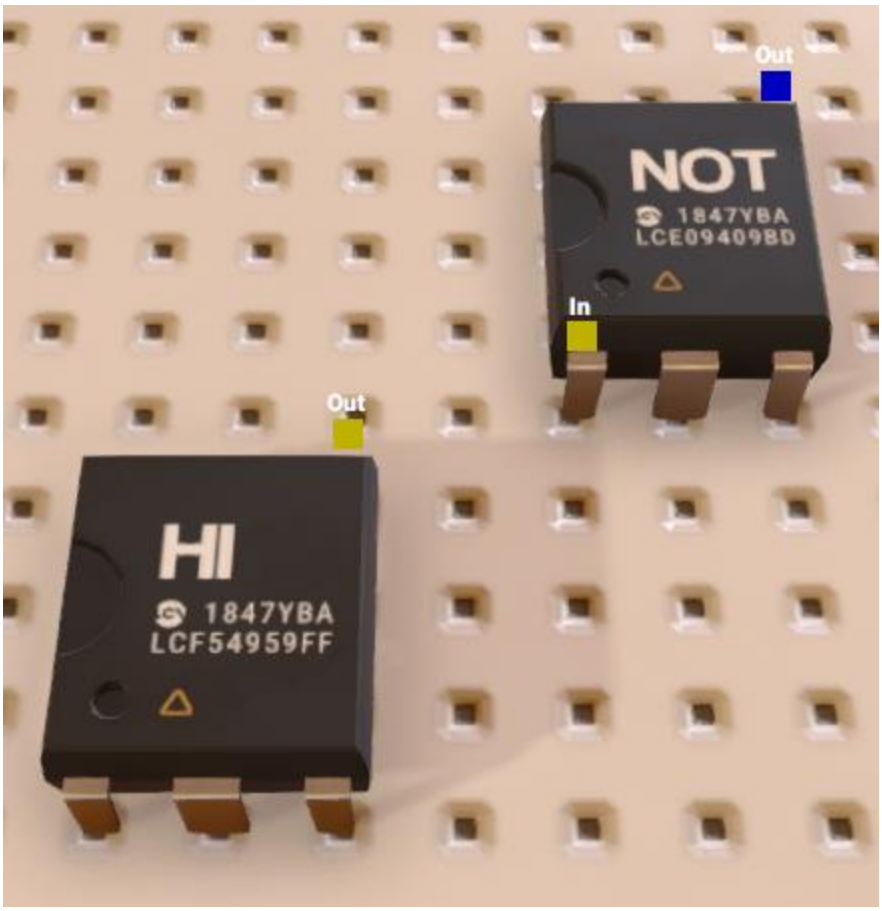

Figure 5: Two Pins Wired Together, as Denoted by Their Shared Color

Presently there is no visual indication of a connection between pins. Although this isn't strictly necessary for testing the behavior of the pin connections it expects a great deal of familiarity with the application from a user, and clearly is not the ideal presentation. Present work is directed at adding a visual line between chip pins to identify connections. In the future we would also like to refine the appearance of the pin icons to be both more legible and more thematically inline with other user interface elements.

Controlling the Breadboard

Our current test board is small enough that it can be viewed in a single screen. However, we do not anticipate this being the case for future boards, nor can we be sure of the resolution of every device that may run the application. To insure users are always able to view any portion of the board on any sized screen a drag gesture was implemented that moves the board across the screen. This gesture is like ones found in applications such as Google Maps, where the position under the cursor when the mouse is depressed remains under the cursor for the duration of the drag.

In actuality the board remains static and the position of the virtual camera is moved instead. Mathematically this is an unimportant distinction, but it simplifies the work involved in managing board coordinates and chip positions upon the board. The vector used to move the camera is calculated by shooting a ray through the mouse cursor and calculating where it intersects the board each frame the mouse button is depressed and taking the difference of this value with the cursor's position at the start of the drag.

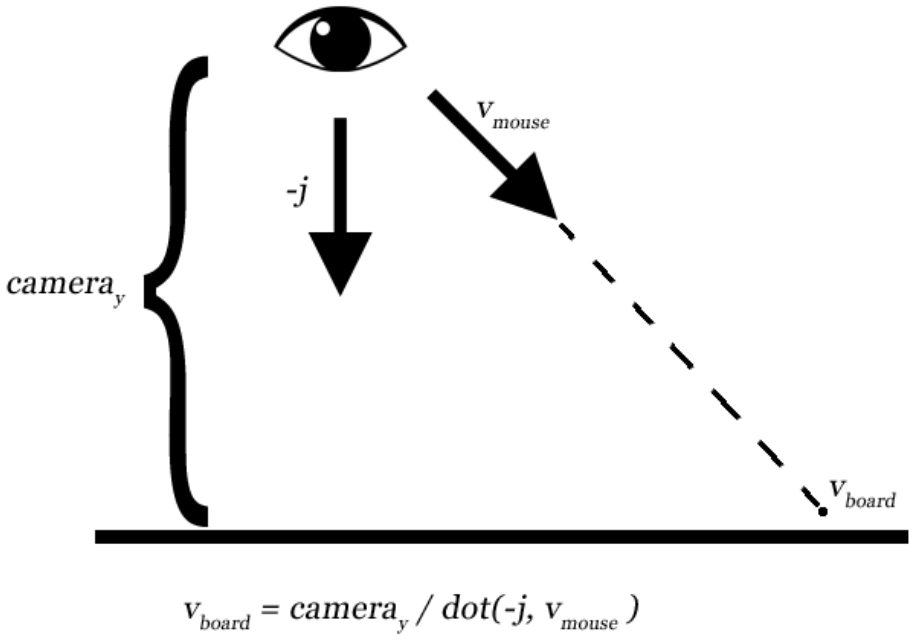

Figure 6: Calculating Mouse Position on Board

Since other interactions require clicking objects on the board, namely chips and pin icons, it's important that small movements of the mouse during a click are correctly interpreted as clicks and not very tiny drags to avoid frustrating users. To this end a "jitter threshold" is defined; if the magnitude of the calculated camera offset vector is below the jitter threshold when the mouse button is released the gesture is treated as a click and not a drag.

Lastly, users may scroll their mouse wheel to dolly the virtual camera in and out, effectively increasing the scale of the board on their screen. This has the practical benefit of letting users on smaller screens dolly out to see the totality of their circuit then dolly in to work on a specific portion of it. There's also a "fun factor" involved in zooming in on a chip and examining it up-close (a task made difficult in reality by the miniscule size of physical microchips). To encourage players to explore their circuits up close fine details were added to the chips, such as mock batch numbers and fingerprint markings.

Conclusion

The Low Carb project was begun with the intent of creating a puzzle game that could be used to instruct those outside the disciplines of computer science and electrical engineering in the fundamental mechanisms underlying the operation of modern microcomputers. Although at this juncture such a game does not exist we have described herein the initial version of a framework that we believe could be used to build such a game. Until usability testing can be performed we can offer no other evidence of success than that which we have provided in these pages. Our own testing of the product has been a quite enjoyable experience, even in its current rudimentary form, which gives us hope that with further refinement a genuinely valuable product can be created.

A great deal was learned from the work on this project. The biggest lesson was in regards to properly scoping a project given a tight timeline (something we failed to do initially). Other valuable lessons included gaining a great deal of experience in time management and work prioritization, as well as learning the value of delivering vertical slices of a product as early as possible. Delivering user interface functionality so late in the project was an experience this author would like to avoid repeating.

We believe the future of the Low Carb project is bright. At this point we could proceed in a number of directions. We have a solid framework in place which can be refined and then built upon to create the originally proposed game. In addition, the codebase can be open sourced to see if others would care to contribute to its further development. One idea that's particularly intriguing is porting the work that's been done thus far to Javascript and WebGL and realigning the project as a web application, accessible from almost any device with no need for an

installation. This would truly satisfy our original goal of accessibility, but further investigation into the technical feasibility of this is necessary.

To conclude on a personal note: working on Low Carb has been an incredibly challenging and rewarding experience. It's certainly pushed me beyond what I knew I was capable of as a developer and as a project organizer. It's been a busy and tiring eight weeks but the end result is something I can genuinely say I'm quite proud of and hope I will have opportunities to continue to develop in the future.

REFERENCES

About SHENZHEN I/O. (n.d.). Retrieved from http://www.zachtronics.com/shenzhen-io/

Arduino - Home. (n.d.). Retrieved from https://www.arduino.cc/E. (2017, August 04).

Ben Heck's Logic Gate board game: The finale. Retrieved from https://www.engadget.com/2017/08/13/ben-heck-s-logic-gate-board-game-the-finale/

Bring ideas to life with free online Arduino simulator and PCB apps | Autodesk Circuits. (n.d.). Retrieved from https://circuits.io/Elliott, R. (2017, December 1).

Craig, D. C. (1991). Circuit Description and Elementary Hierarchical Circuit Simulation Using C++ and the Object-Oriented Programming Paradigm. Retrieved from http://www.cs.mun.ca/~donald/bsc/

Kehagias, D. (2016). A survey of assignments in undergraduate computer architecture courses. Retrieved from http://online-journals.org/index.php/i-jet/article/download/5776/3989

Redstone Circuits. Retrieved from https://education.minecraft.net/lessons/redstone-circuits/

Nisan, N. (2005). The elements of computing systems: Building a modern computer from first principles. London: The MIT Press.

Patterson, D. A., & Hennessy, J. L. (2018). Computer organization and design The hardware/software interface. Cambridge, MA: Morgan Kaufmann.

Petzold, C. (2015). Code: The hidden language of computer hardware and software. Redmond, WA: Microsoft Press.

Savage, P. (2015, January 7). Minecraft player creates word processor out of redstone. Retrieved from https://www.pcgamer.com/minecraft-player-creates-word-processor-out-of-redstone/

Staalduinen, R. V. (2016, April 8). Circuit Scramble. Retrieved from http://suborbitalgames.com/?p=142

Teach, Learn, and Make with Raspberry Pi. (n.d.). Retrieved from https://www.raspberrypi.org/

APPENDIX A: LIST OF TABLES & FIGURES

APPENDIX B: UNUSED USABILITY SURVEY (TECHNICAL)

1. Please rate your prior familiarity with each of the following on a scale from 1 to 5, 1 being no experience and 5 being a professional level of experience.

   Electrical Engineering: ___

   Circuit Design: ___

   Hardware Simulators: ___

   Software Engineering: ___

2. Please identify the most complex software applications you're able to use confidently, such as Computer Aided Engineering software, 3D modeling software, simulation software, etc.

   _____

   _____

   _____

3. Please note any elements of the user interface (buttons, panels, etc.) you found unclear.

   _____

   _____

4. Please note any function in the application the behaved differently than expected. Please include the expected as well as the actual behavior.

   _____

   _____

APPENDIX C: UNUSED USABILITY SURVEY (NON-TECHNICAL)

1. Please identify your prior experience in computer architecture, if any.

   _____

2. Please list your favorite puzzle games, if any. Include computer and video games, tablet or mobile games and board games.

   _____

3. Please enumerate the highest puzzle completed: _____

4. Please note those puzzles that proved the most difficult to complete.

   _____

   _____

5. Please rate your confidence in your understanding of the following topics on a scale from 1 to 5, 1 being no confidence and 5 being absolute confidence.

   Electrical Current and Circuit Design: ___

   NOT, AND, OR and XOR Gates: ___

   Multiplexors and Demultiplexors: ___

   Bitwise Encoding: ___

   Boolean Arithmetic and Adder Chips: ___

   Arithmetic Logic Units: ___

   Bitwise Manipulation: ___

APPENDIX D: TEAM MEMBERS

Aaron Gordon Responsible for all work, including proposal, design, implementation, testing, artwork and final report.

APPENDIX E: SOFTWARE LICENSES

Adobe Photoshop © 2018 Adobe. Licensed for commercial use. Full license available at https://www.adobe.com/legal/terms.html

Autodesk Maya © 2018 Autodesk. Free for educational, non-commercial use. Full license available at https://www.autodesk.com/company/legal-notices-trademarks/software-license-agreements/educational-licensees-additional-terms

Coolvetica Software Font by Typodermic Fonts. Free for commercial, desktop use. Full license available at https://www.dafont.com/coolvetica.font

IBM Plex Mono Software Font by Mike Abbink. Free for commercial use. Released under the Open Font License: http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL_web

Machinations Software Font by Darrell Flood. Free for non-commercial use. Full license available at https://www.dafont.com/machinations.font

Pixellari Software Font by Zacchary Dempsey-Plante. Free for commercial use. Full license available at https://www.dafont.com/pixellari.font

Roboto Software Font by Christian Robertson. Free for commercial use. Released under the Apache license: http://www.apache.org/licenses/LICENSE-2.0

Unity 3D © 2018 Unity Technologies. Free for commercial use to organizations grossing under $100,000 annually. Full license available at https://unity3d.com/legal/terms-of-service