FPGAs4Kids
Spring 2015
Final Report

Kaitlin Huben

May 13, 2015

Abstract

This report details work I have done on the FPGAs4Kids project during the Spring 2015 semester. It is split into four main sections: motivation and objective for the project (including a recap of Fall 2014 work), user interface and gameplay, code internals, and a roadmap for future work.

Contents

1

Motivation and Objective

1.1 Introduction

The FPGAs4Kids project aims to teach children as young as seven years basic concepts in Boolean logic and digital circuits. During the 2014-2015 academic year, we focused on developing software for a game and free development environment that could eventually be connected to an actual FPGA.

1.2 Overview of Current Technologies

Existing programming technologies use a variety of interfaces to teach software and hardware concepts. Most tools (intuitively) use the corresponding interface: hardware to teach and model hardware, software to teach and model software. Some of the existing technologies are mapped out below, without distinguishing between intended age group:

Programming Technologies		**Concepts**	
		Software	Hardware
Interface	Software	Q1 Codecademy [5] Code.org [4] Kids Ruby [12] MIT App Inventor [18] Alice [1] Scratch [23] Lego Mindstorms [14]	Q2 Logisim [17] Modkit [19] Rocky's Boots [21]
	Hardware	Q3 Raspberry-Pi [22]	Q4 LightUp [15] littleBits [16] Play-i [20] Raspberry-Pi (overlap with Q3)

Table 1.1: Sample of existing instructional software and hardware technologies

It seems the most popular way to teach programming concepts is software-to-teach-software (Q1). Software is cheap to distribute, if not to develop, and compilers are well-developed

tools with which to check code development. There is no danger of misplacing small parts or having to purchase many potentially expensive hardware devices. Q1 technologies also seem to have the most development intended for teaching children, with copious "fun" technologies like Codecademy [5], Scratch [23], and Lego Mindstorms [14].

With the intent to focus on hardware instruction for younger audiences, we would find ourselves in either Q2 (software to teach hardware) or Q4 (hardware to teach hardware). Q4, like Q1, seems to be fairly well developed, including for younger ages. There are a few startups like littleBits [16] creating modular hardware pieces that enable younger audiences to quickly learn the basics of hardware. Children can connect hardware with their own hands and observe results without having to dive into the minutiae of logic and low-level design.

Though Q4 technologies have some obvious upsides, such as direct interaction with hardware, we would like to focus on Q2. One longstanding Q2 technology is Logisim, "an educational tool for designing and simulating digital logic circuits" [17]. While usable for students in higher education, it is not a particularly user-friendly or engaging tool for younger audiences. Another Q2 technology, Modkit, provides a visual programming environment for a variety of hardware boards [19]. While usable for competent programmers, it is also not conducive to learning basic hardware (or software) principles.

The Q2 technology most accessible to new users was a product of the 1980s: Rocky's Boots, an educational logic puzzle released in 1982 [21]. Gamers had to link basic logic gates (NOT, AND, OR, etc.) together to wire a boot to kick correct shape and color combinations. Rocky's Boots received rave reviews from educational and gaming communities alike before fading into the wash of new technologies.

Our goal is to take the inspiration behind Rocky's Boots – teaching low-level hardware and logic concepts – to create a new game with expanded aims. Ideally, we would be able to expand the "game" component into a free-form development interface utilizing the same visual programming library and enabling new developers to work on actual hardware, as outlined in section 1.3.

1.3 Objectives

After deciding to work in the software-to-teach-hardware realm (Q2), we outlined everything that we wanted to accomplish. Figure 1.1 shows the overall structure of the project.:

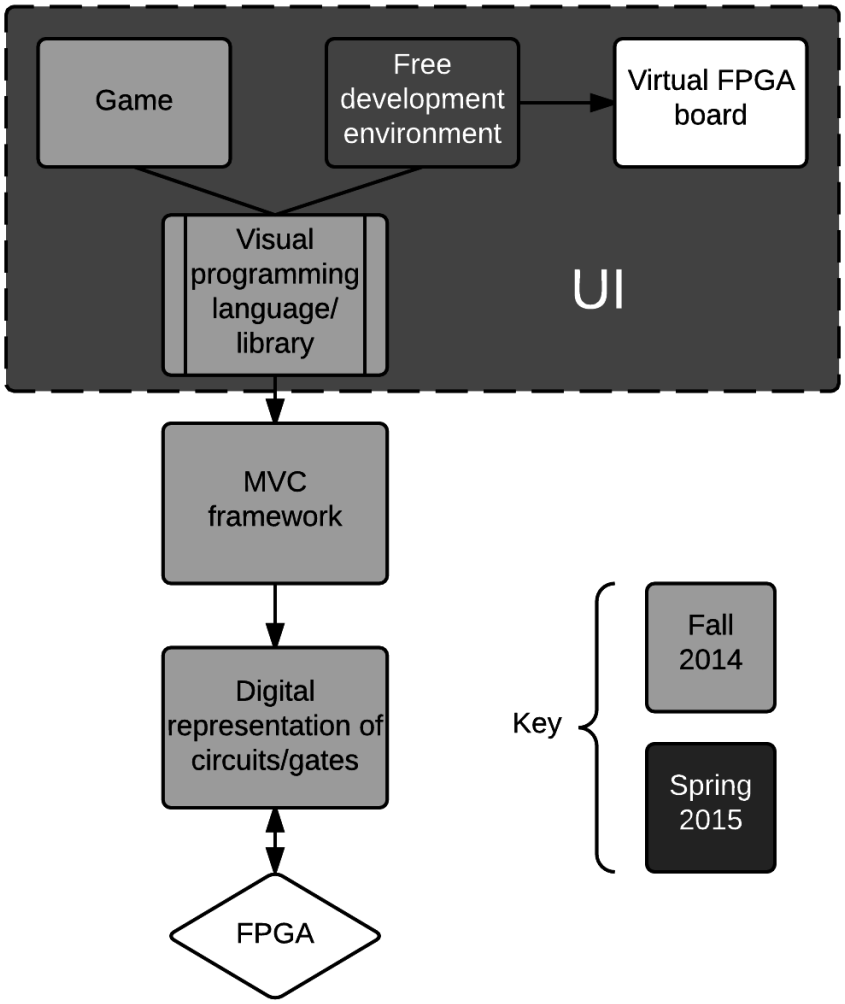

Figure 1.1: Proposed conceptual design of FPGA4Kids project

During the first semester, we focused on building the game aspect of the project. This would provide a fun way for children to learn the basics of Boolean logic and circuits. (Section 1.4 provides more information about work done in Fall 2014.) This semester we continued work on the game while also developing the framework for a free development environment. In the future, we would hope to develop an virtual FPGA board as well, along with a way to translate between circuits developed in the software and an actual FPGA board. Translating between the software's representation of the circuit and a physical FPGA could take a variety of forms; one potential way to do this would be to use XML (used by Logisim) and Verilog as intermediate steps (shown in Figure 1.2).

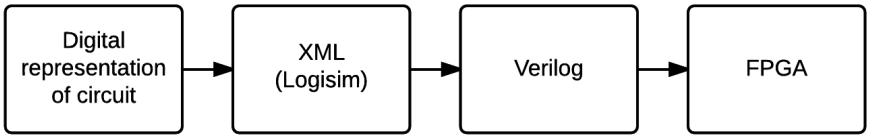

Figure 1.2: Potential translation from software's circuit representation to FPGA

1.4 Fall 2014 Recap

This section summarizes work completed and planned in Fall 2014.

Section 1: Motivation and Objective. This was covered in the previous section of this report.

Section 2: Description of Work Accomplished. This section began with preliminary research on game development and hardware education at all levels. It then described our proposed curriculum, which included Boolean values; NOT, AND, NAND, OR, NOR, XOR, and XNOR gates; possible combinational logic (e.g. encoders, decoders); and possible sequential logic (latches, flip-flops, clocking). We noted here that we wanted our software to have the capability to model arbitrary circuits in addition to simpler game levels, something that we emphasized this semester. The outline of software decisions included our choice to use Elm, and the game design decisions included "Tutorial levels", "Basic levels", and "Advanced levels". This report details many more decisions that have since been made about user interface design and gameplay. The software (re)development subsection outlined what had been accomplished in the fall semester. This included:

- Basic design of MVC framework
- Data model for gates, circuit state, and game state
- Simulation logic for individual gates and arbitrary circuits
- Display logic, including naive circuit routing
- Simple control logic accessible by individual levels
- Almost completed upgrade to Elm v0.14

Section 3: Code Documentation. This section outlined the general architecture and described how to set up a level. Section 3 of this report contains the up-to-date architecture and level set-up information.

Section 4: Future Work. The main items that were started or completed during this semester were completing the upgrade to Elm v0.14 and planning a free development environment. Plans for future work are in relevant sections of this document, as well as in section 4.

2

Gameplay and User Interface

We originally outlined the following types of levels:

- "Tutorial levels," which would introduce the user to basic concepts and gameplay.
- "Basic levels," which would allow the user to manipulate only inputs to a given set of gates to try to gain certain outputs.
- "Advanced levels," which would allow the user to manipulate only gates with a given set of inputs to try to gain certain outputs.

Partway through the semester, we added a fourth type of level category:

- "CAD levels," which would give the user control over combinations of inputs, gates, nets, and outputs.

We currently have some fully functional tutorial and basic levels, and we began the planning, prototyping, and user studies for CAD levels. The following sections describe those current levels and planned CAD levels.

2.1 Current Levels

The framework is mostly in place for tutorial and basic levels. Section 3 describes the single-page architecture and the idiom for setting up individual levels.

2.1.1 Current User Interface and Gameplay

The `MainRunner.elm` file in the `src` directory currently runs a single tutorial-style level containing one NOT gate. The user is first presented with the screen as shown in Figure 2.1.

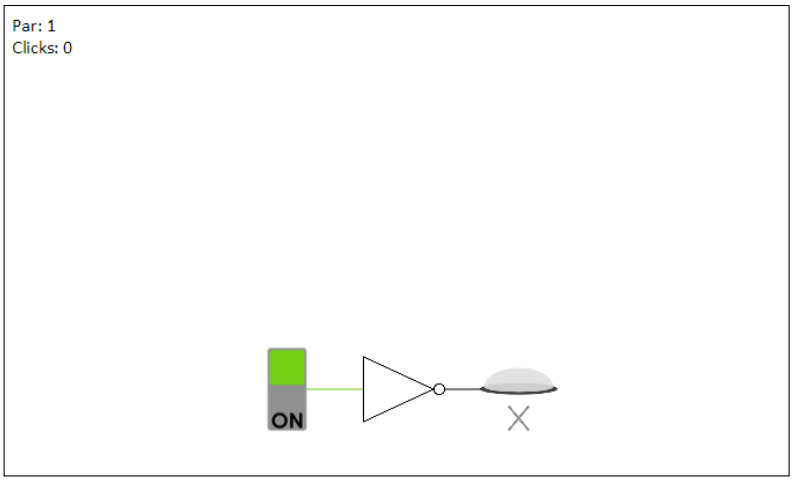

Figure 2.1: Screenshot of the beginning of Level 1.

Figure 2.2 annotates the components of the level. The upper left corner contains level statistics, where "Par" is the minimum number of clicks needed to complete the level and "Clicks" is the number of times the user has clicked on an input. Looking from the left of the circuit, we see the following:

- The input. This is the only thing the user is allowed to change. It is also the only clickable item on the screen; this is emphasized by having the mouse cursor change to a pointer when it hovers over the input. When the user clicks on the input, it changes from on to off (or vice versa).
- The net from the input to the NOT gate. Since the input is on, the net is on and is colored green.
- The NOT gate. It is currently in schematic form.
- The net from the NOT gate to the output. Since the NOT gate has an on input, it has an off output, so the net is colored gray.
- The output. It is currently off (gray).
- The 'X' beneath the output. This indicates that the output is not correct, e.g. the solution involves the output being on instead.

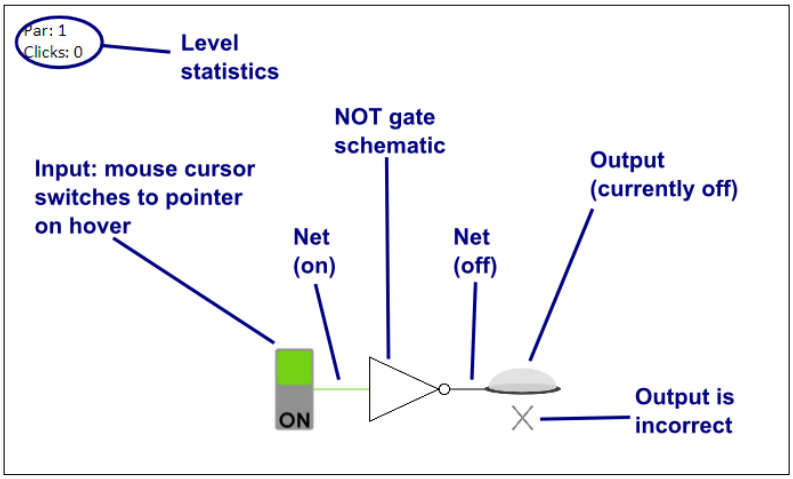

Figure 2.2: Annotated screenshot of the beginning of Level 1.

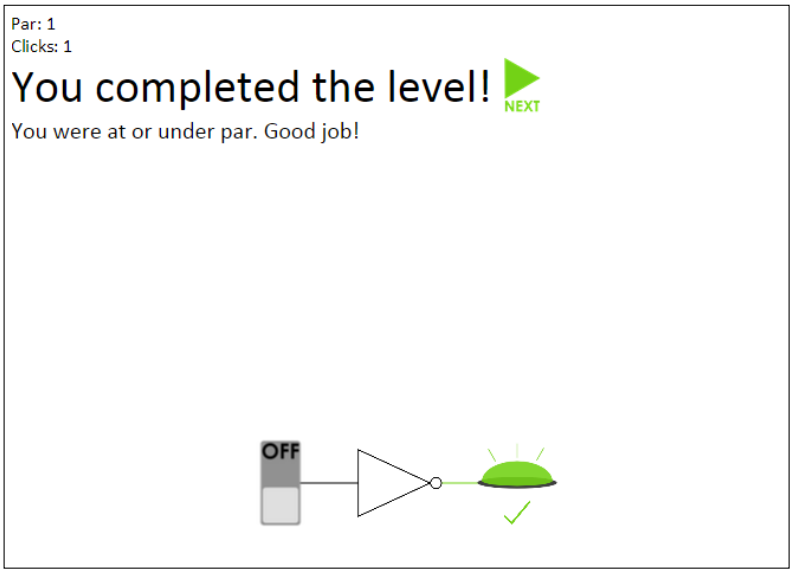

Figure 2.3: Screenshot of the end of Level 1.

After the user clicks on the input, they see the screen shown in Figure 2.3.

Notice the following changes:

- There is a new banner at the top of the screen indicating that the level has been completed. It contains a button to go to the next level and a message indicating whether the user was "on par", e.g. used no more than the minimum number of clicks to complete the level.
- The input has changed from on to off.
- The net from input to NOT gate has changed accordingly.
- The net from the NOT gate has also updated according to the underlying circuit simulation.
- The output is now on (green).

- There is now a green checkmark underneath the output, indicating it is correct.

2.1.2 Future Work: Combining Levels

Currently, the `MainRunner.elm` file only contains Level 1 as shown above. Once the single-page architecture is fully implemented (see section 3.5), the "Next" button will actually take the user to another level.

Four more levels were quickly designed under the multi-page architecture for a brief user study, as outlined in the next section.

2.1.3 Future Work: Integrating User Study Feedback

Figures 2.4 and 2.5 show the 5 levels that were used during user testing. They are still up at `http://kaitlinhuben.github.io/FPGAs4Kids`. (Keep in mind that these levels look slightly different since they are an older version of the architecture.) The user studies yielded quite a bit of valuable feedback, which is broken down by category below.

Overall observations and feedback

- Users heavily emphasized the text directions at the top of each level, often at the expense of paying attention to and retaining what was happening in the circuit. The text in level 5 was particularly confusing. There are many potential fixes here, including having overlays in tutorial levels that bring the content of the directions closer to what is being described, or trying to make the directions more graphically-based.
- The overall goal of each level was typically unclear. The system of Xs and checkmarks was not immediately intuitive for users. A possible fix is having the "solution" (the desired outputs) displayed next to the current outputs on tutorial levels.
- Users seemed to get comfortable with the tutorial levels pretty quickly. As described before, this sometimes meant they didn't pay attention to what each gate was actually doing.
- The "Next" buton was frequently overlooked, especially in the first level.

Users with prior electrical engineering experience

- Users comfortable with circuit schematics tended to skim the text directions very quickly.
- Users didn't have much trouble transitioning from tutorial levels to basic levels.

Users with no prior electrical engineering experience

- Users placed heavy emphasis on the text directions.
- Users had trouble retaining information between levels - tutorial levels did not seem to have their desired effect.
- Users had trouble with the distinction between inputs, gates, and outputs. These need to be made more intuitive.

- Users were bogged down by terminology (gate, AND, OR, etc). It might be useful to actually label schematics with the name (but keep in mind the game will look entirely different).
- Users felt that Level 4 (the first basic level) was too difficult too quickly after simple tutorial levels. This is easily fixed by creating new levels.

2.1.4 Future Work: Game Imagery

Everything discussed so far has related directly to schematic versions of the game. Ultimately, game play should be fun, especially for children. We brainstormed physical analogies for gates to make gameplay more intuitive. While the overarching theme of the game is still very open, we decided that water is probably the best analogy for flow of electricity through a circuit. Figure 2.6 shows some possible implementations of gates with water, and figure 2.7 shows how they would knit together into a game.

These prototypes were developed very early in the project and would likely change quite a bit based on current architecture. Many of the implementations also needed a fair amount of work; for example, the NOT gate is not particularly intuitive or physically possible. A possible alternative would be a faucet or other thing that can actually be turned on or off.

2.2 Free Development Environment ("CAD levels")

The idea of developing "CAD (Computer-Aided Design)" levels came up relatively late in the project. It was proposed as a single unifying architecture for all types of levels, rather than developing basic, tutorial, and advanced levels independetly from each other and from the free development environment. CAD levels would be based on a fully functional free development environment, where users could put inputs, gates, and outputs from a library anywhere within a build area and connect the components as they see fit.

From CAD levels, it would be easy to add restrictions (or user permissions) on types of components to create different kinds of levels. For example, "basic levels" have typically allowed the user only to change inputs from on to off and vice versa, encouraging them to look at the effect this has on the circuit. This could be viewed as a CAD level where the user is not allowed to move any components, is not allowed to change any nets, and therefore can only change the state of the inputs. While this example may seem somewhat contrived, using this approach makes developing other kinds of levels much easier. Advanced levels could allow users to change nets, pick gates, or fill in gaps with a specific set of gates.

2.2.1 Prototypes and Decisions

Figures 2.8 to 2.11 show initial prototypes developed for CAD levels. Some important things to note and additional ideas:

- The idea of discarding with some sort of trash bin is likely too complicated. All references to the trash bin in the prototypes should be disregarded.
- The library of components can be located either at the left/right or top/bottom of the screen, depending on screen orientation.
- The library of components can show components stacked to give visual feedback for how many components are left.
- It's not clear exactly how the build space should be laid out, e.g. grid, just dots for component placement, or nothing at all. Regardless, components should snap to the underlying grid, which should restrict users to Manhattan geometry.
- Components should be placed by clicking rather than by drag-and-drop, as described in [11]: "Within the context of the puzzle-solving game, the children solved significantly fewer puzzles, and they were less motivated using the version that utilized a drag-and-drop interaction style as compared to the version that utilized a point-and-click interaction style."
- The concept of a "don't care" wire or input does not yet exist and will probably be a useful addition to the CAD levels.

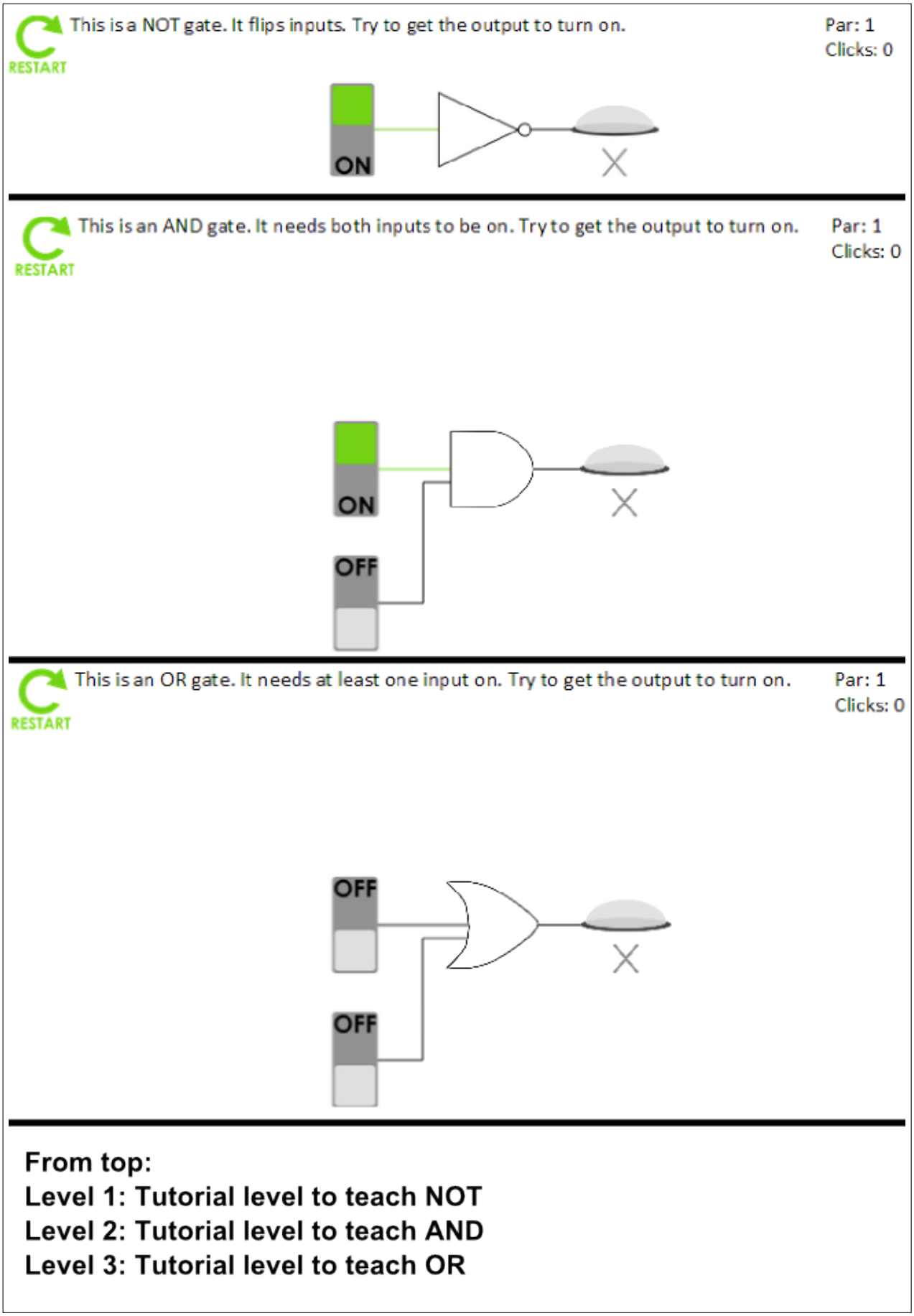

Figure 2.4: First three levels used in testing.

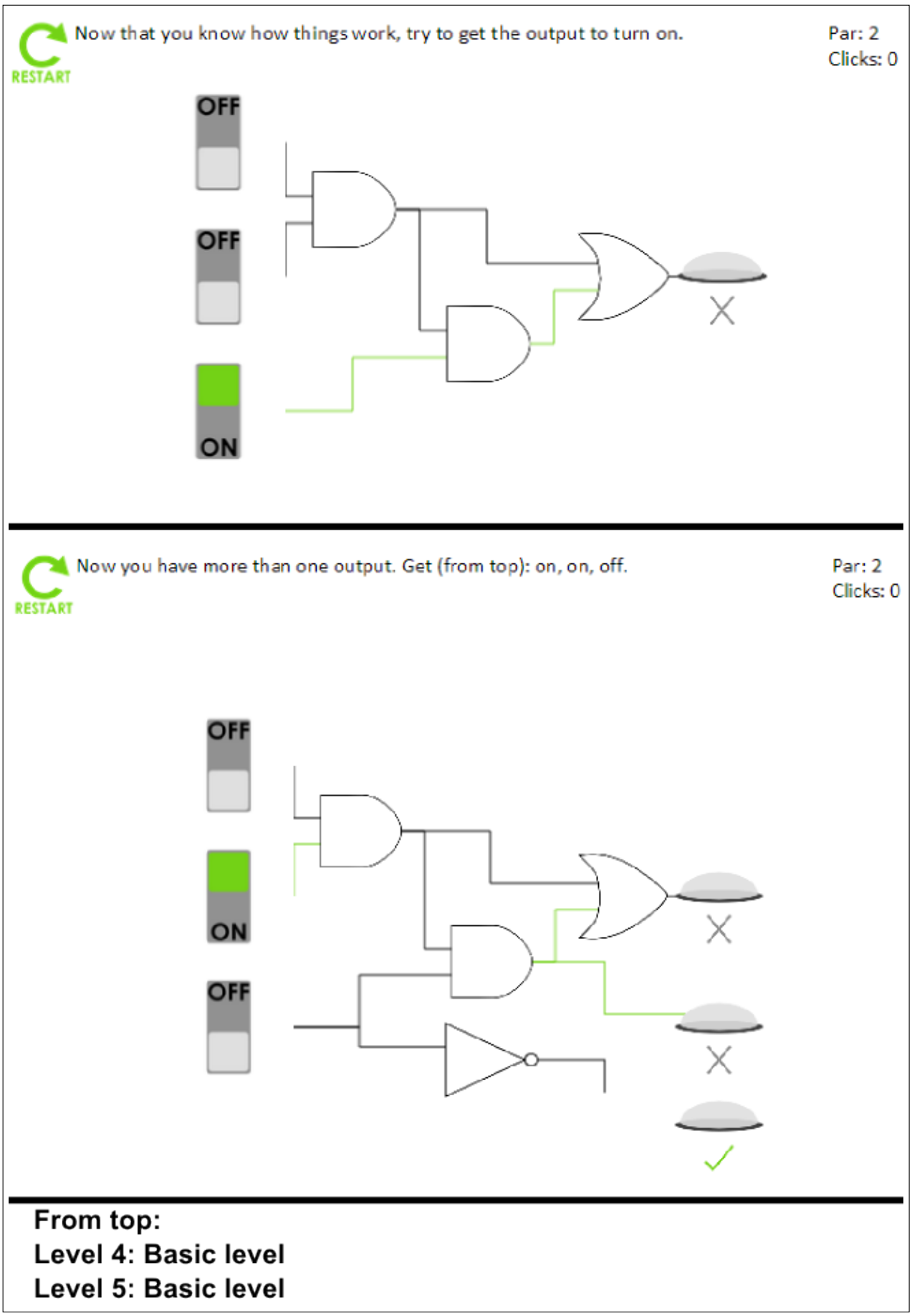

Figure 2.5: First three levels used in testing.

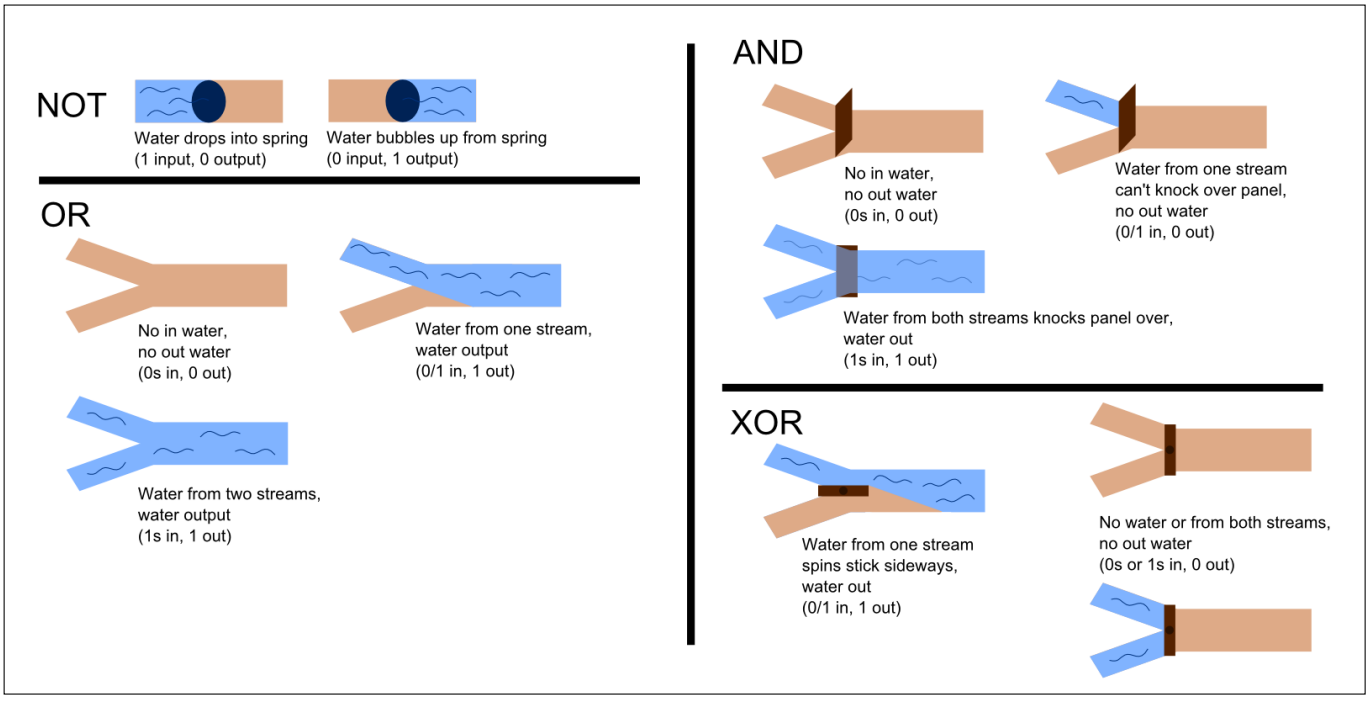

Figure 2.6: Possible water visualizations of gates.

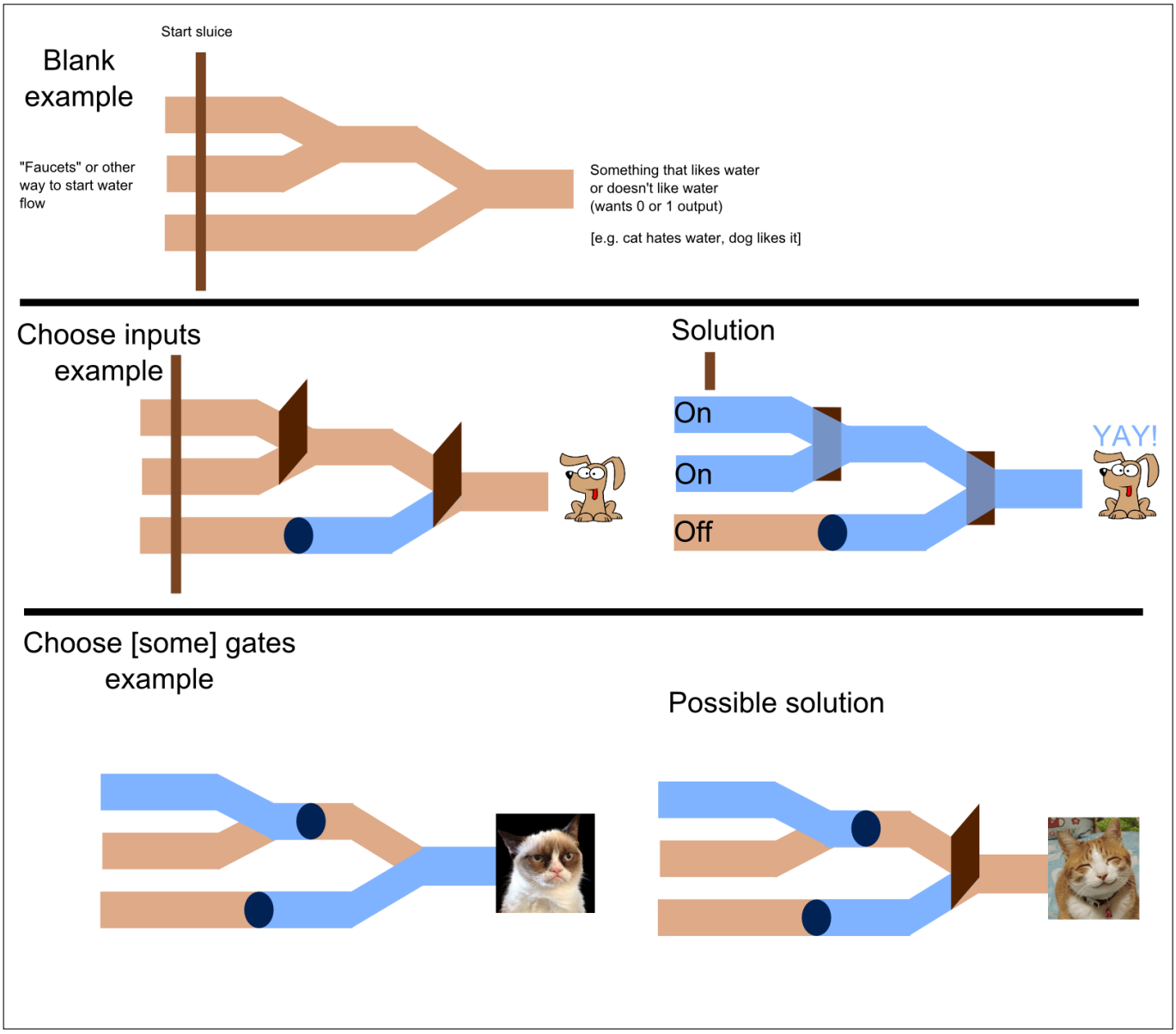

Figure 2.7: Gameplay with water gates.

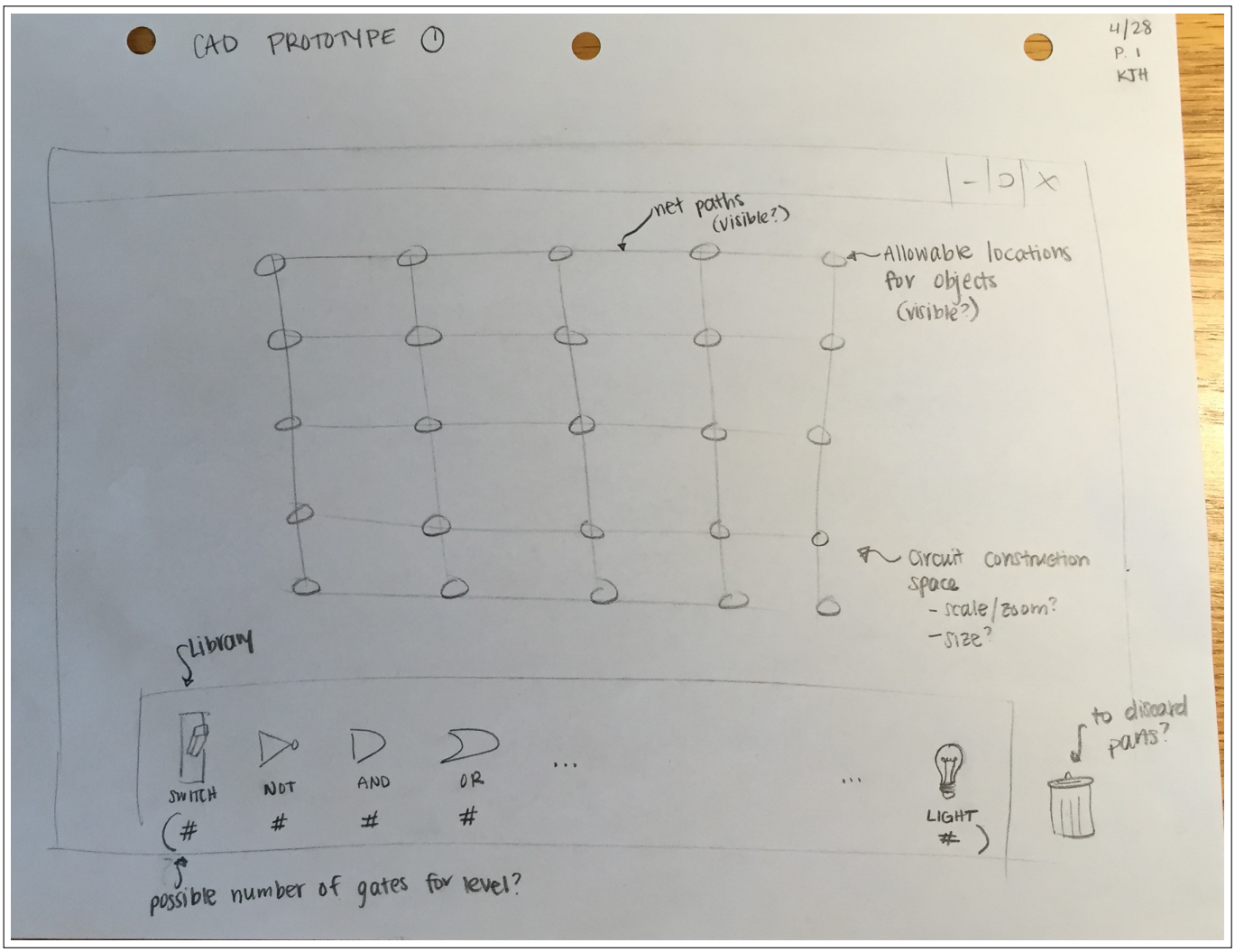

Figure 2.8: General CAD prototype with library at bottom of screen.

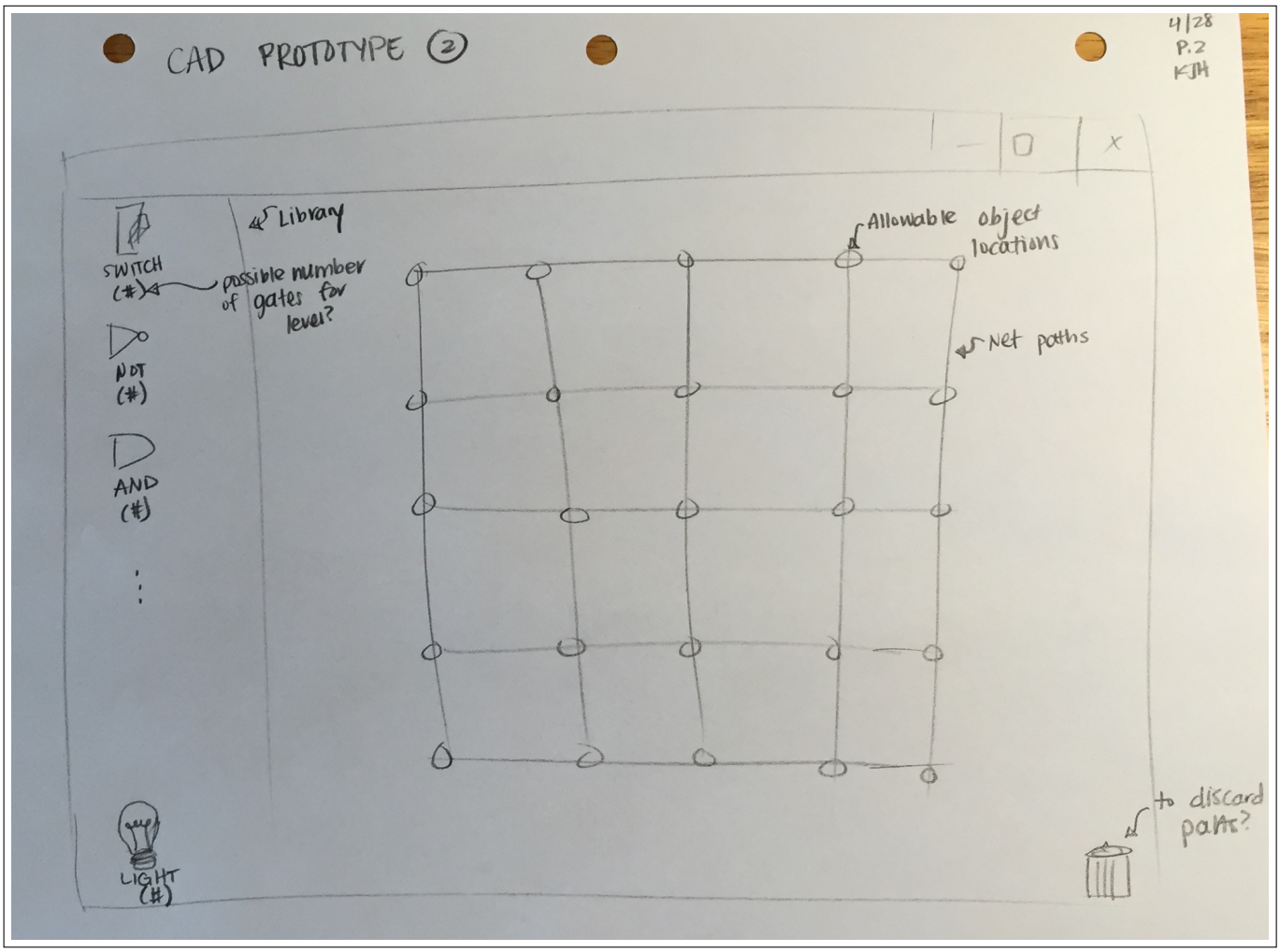

Figure 2.9: General CAD prototype with library at left of screen.

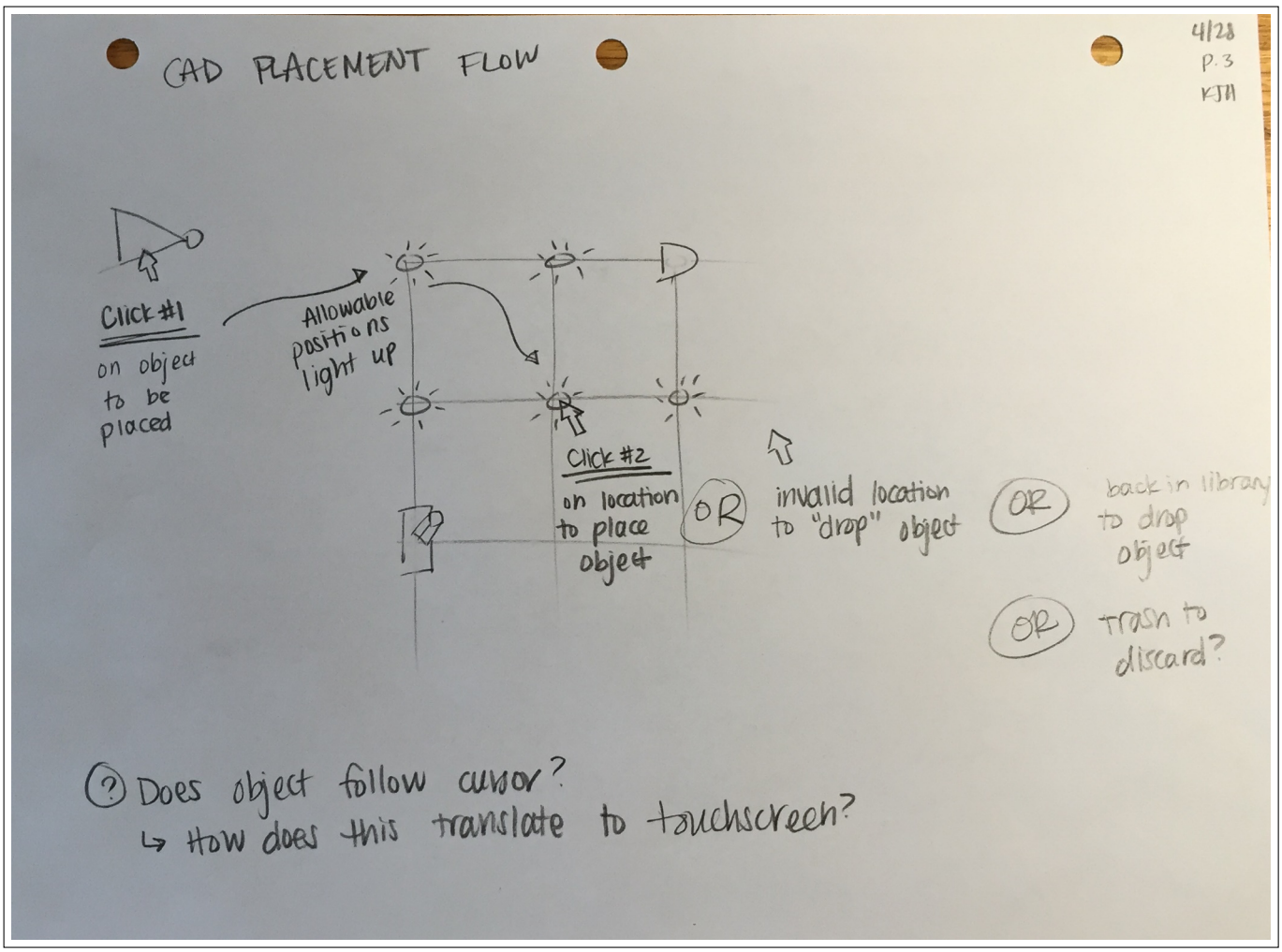

Figure 2.10: Placement flow for CAD level, putting a component on the circuit.

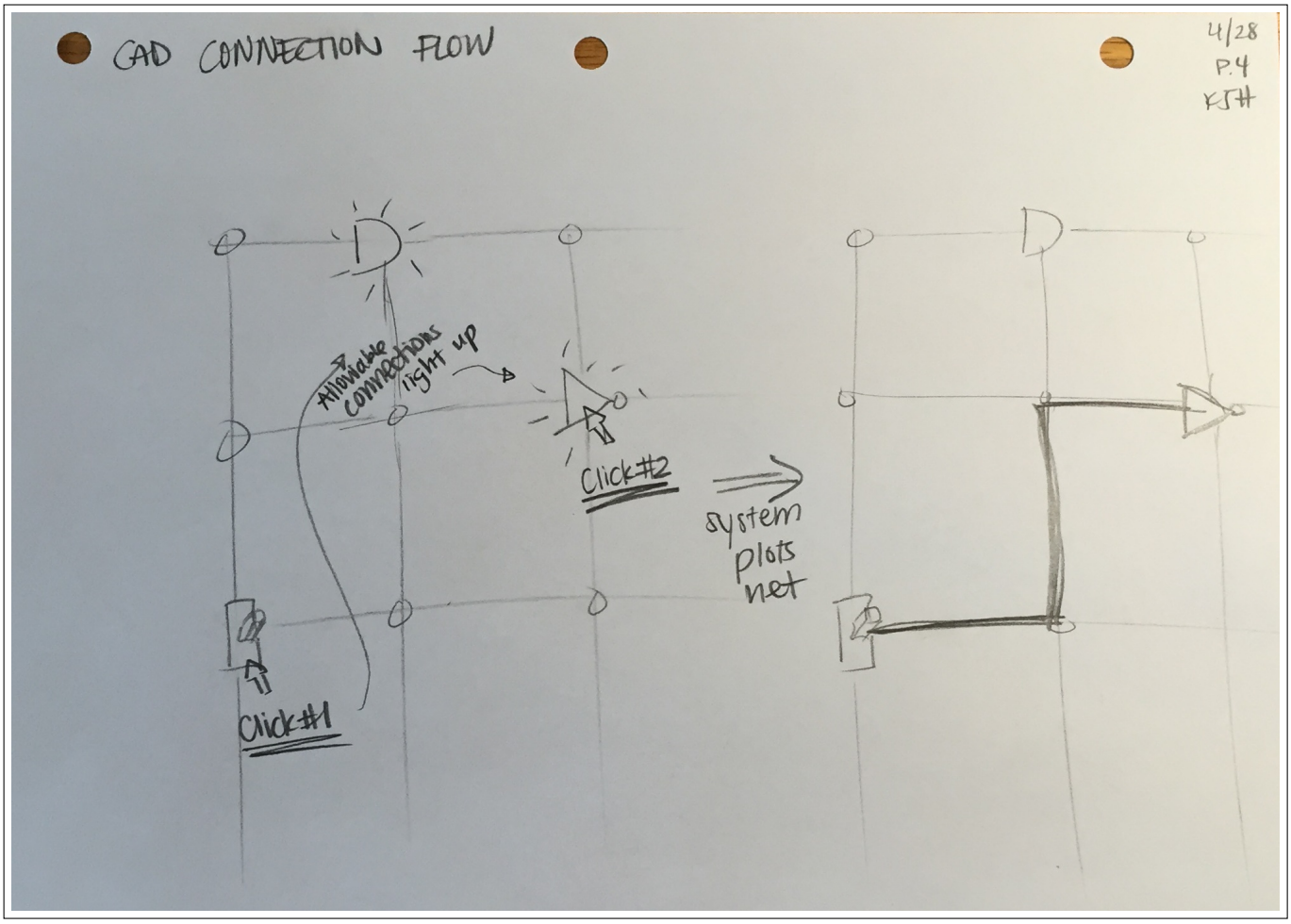

Figure 2.11: Connection flow for CAD level, connecting two components.

3

Code Documentation

3.1 GitHub Repository

All code is located at `https://github.com/kaitlinhuben/FPGAs4Kids`.

3.2 Summary of Major Changes

3.2.1 Version 0.13 versus 0.14 versus 0.15

At the end of the Fall 2014 semester, the upgrade from 0.13 to 0.14 was mostly complete; it was finished early in the Spring 2015 semester. In April 2015, Elm v0.15 was released. After realizing many modules were not compatible with this upgrade, the code was left compatible with v0.14.

3.2.2 Multi-Page versus Single-Page Architecture

The original multi-page architecture from the fall semester is still in the code repository, in the `original_architecture` folder. Under this paradigm, each level became its own HTML page. This semester we moved toward a single-page architecture, like the Maze example (`https://github.com/Dobiasd/Maze`). Instead of multiple HTML pages, there is a single HTML page that repaints the page for each level, popping levels off of a stack as the user progresses. The code does not currently actually switch between levels, but that functionality should be fairly easy to implement using Maze as an example.

3.2.3 Minimizing Instantiation Requirements

Previously, creating a new level required creating many redundant data structures manually. The `initGameState` function (formerly `instantiateGameState`) has been expanded to take care of most of that work, meaning much less information is needed to set up a level.

3.2.4 Removing `fps`

Previous code had used the `fps` function to trigger the page to repaint 25 times per second. The code now only updates the page when an event has actually occurred (e.g. user clicks on input).

3.3 General Architecture Overview

3.3.1 Modules

The code for the game is split into a few different modules, following an MVC framework. (For a more detailed discussion of how the modules interact within the MVC framework, see section 3.3.2.)

- Model
  - Model: This module contains all of the types and type aliases needed to model the game. This includes
    - `GateInfo`: all information needed to create a gate in a level
    - `Level`: all information about a single level, including a `List` of `GateInfo`, one for each gate
    - `UserInput`: all inputs provided from user, like input gate on/off choices and mouse position
    - `GameMode`: whether the user is in the game or schematic UI
    - `GameState`: all current information about the game, like circuit state and game mode
    - `CircuitState`: current state of the circuit in the level
    - `InputsState`: current state of all inputs to circuit in the level
    - `GateType`: whether the gate is an input, output, or other gate
    - `Gate`: to model a single gate

    It also includes logic functions for each type of gate and `String` constants with the path to gate images.
    Lastly, with the upgrade to v0.14 that does not support `getOrFail`, this module contains `get` methods that return default `failed` instances upon failure.
  - CircuitFunctions: This module contains all functions needed to update the `GameState` by simulating the circuit. The `Controller.Controller` module accesses gate simulation with the `updateGameState` method.
- View
  - View: This module contains functions needed to display the game to the user. The `Controller.Controller` method accesses display functionality with the `display` method.
  - Nets: This module is used by the View module; it contains functions needed to draw nets.
- Controller
  - Controller: This module contains all of the functions needed to collect input from the user and update the `GameState` appropriately. Any level accesses the main game functionality with the `mainDriver` method.
  - InstantiationHelper: This module contains all of the functions to instantiate a `GameState` with minimal work from each level. Any level accesses the main instantiation functionality with the `initGameState` method.

3.3.2 MVC Framework

The architecture loosely follows an MVC design. Figure 3.1 below shows interactions between modules and levels. Note that `MainRunner` only needs to import `Model.Model` for type definitions, `Controller.InstantiationHelper` to automate `GameState` setup, and `Controller.Controller` to actually run the level.

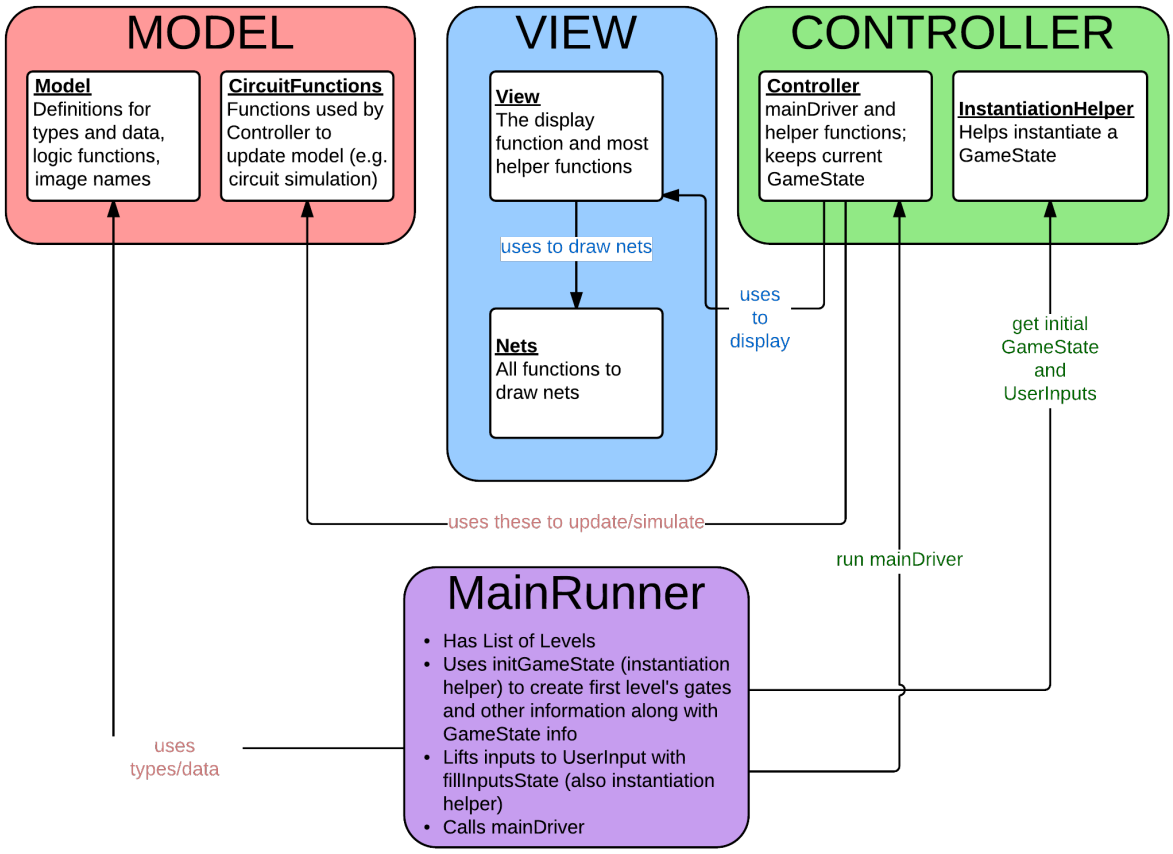

Figure 3.1: How `MainRunner` interacts with the MVC framework

The `Controller.Controller` module contains the `mainDriver` code, which actually runs the game. Figure 3.2 shows the interactions between `mainDriver` and its helper functions.

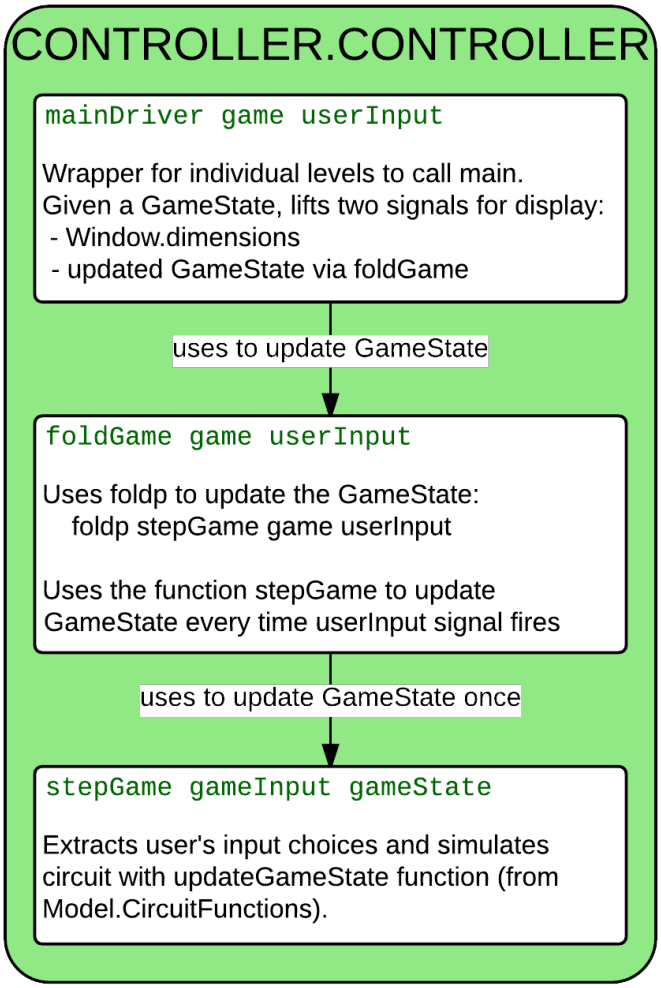

Figure 3.2: How the `mainDriver` function works

3.4 Setting up a level

The current idiom for setting up a game level is much simpler than the prior one. There are 3 steps: create the list of level information, instantiate the game state and user inputs, and run the main driver.

3.4.1 Create the `List` of `Levels`

There are seven pieces of information required to set up a level.

1. name: The name of the level.
2. solution: A `List` of `(String, Bool)` pairs. In each pair, the `String` should be the name of an output gate and the `Bool` should be its desired value.
3. nextLevel: The name of the next level.

4. par: How many clicks it takes to complete this level.
5. directions: Text directions for the user.
6. gateInfo: A `List` of `GateInfo`, one for each gate in the level.
7. channels: A `List` of `(String, Channel Bool)` pairs. In each pair, the `String` should be the name of an input gate and the `Channel Bool` should be an input channel from which to read user inputs.

Listing 3.1 shows the current `List` of one `Level` contained in `MainRunner`.

Each `GateInfo` record requires the following:

- name: Name of the gate.
- gateType: InputGate, NormalGate, or OutputGate.
- status: Whether the gate starts the level on or off.
- inputs: A `List` of inputs to the gate (if any).
- logic: A `String` containing the name of the function to use to simulate internally. (This should be changed to the function itself).
- location: A pair of `Float`s indicating the location of the center of the gate with respect to the center of the circuit container.
- imgSize: A pair of `Int`s indicating the size of the gate image.

Listing 3.1: The current `List` (containing only one `Level`) in the `MainRunner` file.

```
levels : List Level
levels =
  [
    {
      name = "Level_01"
    , solution = [("outputGate", True)]
    , nextLevel = "Level_02"
    , par = 1
    , directions = "Testing directions"
    , gateInfo = [
        { name = "inputGate"
        , gateType = InputGate
        , status = True
        , inputs = []
        , logic = "inputLogic"
        , location = (-100,0)
        , imgSize = (75,75)
        },
        { name = "notGate"
        , gateType = NormalGate
        , status = False
        , inputs = ["inputGate"]
```

```
        , logic = "notLogic"
        , location = (0,0)
        , imgSize = (75,75)
        },
        { name = "outputGate"
        , gateType = OutputGate
        , status = False
        , inputs = ["notGate"]
        , logic = "outputLogic"
        , location = (100,0)
        , imgSize = (75,75)
        }
      ]
    , channels = [("inputGate", channel True)]
    }
  ]
```

3.4.2 Instantiate the `GameState` and user inputs

The `initGameState` function (in the `InstantiationHelper` module) takes a `Level` as a parameter and returns the corresponding `GameState`. Note that once the single-page architecture is fully implemented, the next level to be played will always be at the head of the list of levels.

Listing 3.2: Instantiating the `GameState` with the `initGameState` function.

```
gameState : GameState
gameState = initGameState (List.head levels)
```

The `fillInputsState` function creates a `Signal (InputsState)` for a level given its channels. It also uses the helper function `getEmptySignalInputsState` to grab an empty inputs state to which to add the level's channels.

Listing 3.3: Instantiating the `UserInput` with the `fillInputsState` function.

```
firstLevel = List.head levels

userInputs : Signal (InputsState)
userInputs = fillInputsState firstLevel.channels
   getEmptySignalInputsState
```

3.4.3 Run `main`

Once you have the `GameState` and `UserInput`, you run the `Controller`'s `mainDriver` method. Since the `mainDriver` function expects a `UserInput` record, the `map` function is used to map the `Signal (InputsState)` into the `UserInput`.

Listing 3.4: Run everything.

```
main : Signal Element
main = mainDriver gameState (map UserInput userInputs)
```

As long as everything is set up correctly, this should run without trouble.

3.5 Future Work: Code Fixes and Additions

As discussed previously, the switch to single-page architecture needs to be fully implemented so that a user can actually transition between levels. There are a few other small things in the code marked with `TODO`, like adding more gate categories (instead of just `InputGate, NormalGate, OutputGate`) and fixing a bug in the nets drawing function. (Neither of these are critical, which is why they're still `TODO`s!) The concept of a don't care wire should be introduced. Perhaps most importantly, a non-naive wire routing algorithm is critical, especially if later CAD levels are to work properly.

4

Future Work

There is much work to be done before the code it is a testable game. Proposed future work on the FPGAs4Kids project includes the following:

- Complete the single-page architecture as described throughout this report
- Address user feedback as outlined in section 2.1.3
- Enhance and implement game imagery as outlined in section 2.1.4
- Complete the free development environment as described in section 2.2
- Complete future coding work as outlined in section 3.5
- Add more touchscreen responsiveness
- Come up with a name for the game
- Test on multiple devices (computers [all browsers], phones, tablets, etc.)
- Create a virtual FPGA board
- Translate from Elm representation of gates to actual FPGA

Bibliography

[1] Alice, http://www.alice.org/.

[2] Begosso, Luiz Carlos and Priscila Rodrigues da Silva, "Teaching computer programming: A practical review", *2013 IEEE Frontiers in Education Conference (FIE)*, (2013): 508-510. doi:10.1109/FIE.2013.6684875.

[3] Bencheva, N., N. Kostadinov, and Y. Ruseva, "On Teaching Hardware/-Software Co-design using FPGA", *Electronics and Electrical Engineering* 102, no.6 (2010): 91-94. http://www.ee.ktu.lt/journal/2010/6/23__ISSN_1392-1215_On%20Teaching%20Hardware%20Software%20Co-design%20using%20FPGA.pdf, accessed September 16, 2014.

[4] Code.org, http://code.org/.

[5] Codecademy, http://www.codecademy.com/.

[6] Doherty, Liam and Vive Kumar, "Teaching Programming Through Games", *International Workshop on Technology for Education (TAE)*, (2009): 111-113. doi:10.1109/T4E.2009.5314120.

[7] Elm, http://elm-lang.org/.

[8] Elm Learn, http://elm-lang.org/Learn.elm.

[9] Hall, Tyson S. and James O. Hamblen, "Using FPGAs to Simulate and Implement Digital Design Systems in the Classroom", *2006 ASEE Southeast Section Conference*, (2006). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.8372&rep=rep1&type=pdf, accessed September 16, 2014.

[10] Haskell, Richard E. and Darrin M. Hanna, "What are the Fundamentals of Digital Design?", *Proceedings of the 2005 ASEE North Central Conference, American Society for Engineering Education*, (2005). http://www.richardhaskell.com/files/Haskell_Hanna_ASEE2005_paper.pdf, accessed September 16, 2014.

[11] Inkpen, Kori M., "Drag-and-Drop versus Point-and-Click Mouse Interaction Styles for Children", *ACM Transactions on Computer-Human Interaction*, 8, no. 4 (March 2001). http://www.cise.ufl.edu/research/lok/teaching/hci-f07/papers/inkpen.pdf, accessed April 23, 2015.

[12] KidsRuby, http://kidsruby.com/about.

[13] Krathwohl, David R., "A Revision of Bloom's Taxonomy: An Overview." *Theory into Practice* 41, no. 4 (Autumn 2002): 212-218. JSTOR (1477405): http://www.jstor.org/stable/1477405.

[14] LEGO Mindstorms, http://www.lego.com/en-us/mindstorms/.

[15] LightUp, http://www.lightup.io/.

[16] littleBits, http://littlebits.cc/.

[17] Logisim, http://www.cburch.com/logisim/.

[18] MIT App Inventor, http://appinventor.mit.edu/explore/.

[19] Modkit, http://www.modkit.com/.

[20] Play-i, https://www.makewonder.com/.

[21] Rocky's Boots, http://en.wikipedia.org/wiki/Rocky's_Boots.

[22] Raspberry-Pi, http://www.raspberrypi.org/.

[23] Scratch, http://scratch.mit.edu/about/.