Logic Circuit Workbench: GatePlay

Greg O'Connor
Computer Science, University of Oxford

May 20, 2014

Abstract

GatePlay is a web application to build and simulate logic circuits from within the web browser, without the need for installation or proprietary plug-ins. It is designed to couple a simple and aesthetically pleasing Graphical User Interface with an efficient and accurate simulation.

Contents

Chapter 1

Introduction

1.1 Motivation

Logic circuits underpin practically every aspect of modern life in the form of computer chips. Therefore it is important that people have the opportunity to learn about and understand logic circuits.

I believe that one of the best ways to learn about logic circuits is to build them, and that the most accessible way of doing that is by using a simulated workbench. The workbench should be very simple to use, but without simplifying the simulation to the point of being unhelpful.

The best workbenches I could find on the Internet were Logic.ly[1] and CircuitLab[2]. Even they had problems such as overly simplistic simulations, overly complicated user interfaces, or requiring the need for trusting or installing their program.

1.2 Overview of GatePlay

The main interface of GatePlay can be seen in figure 1.2. The labelled regions are:

1 The **top bar** which has buttons to download the workbench as an image and to start simulating the circuit.When simulating, the top bar has additional controls such as starting, restarting, and pausing the simulation.

2 The **left bar** which has sliding panels which contain components available to build circuits with. The user simply drags a gate from the left bar onto the workbench to add it to the circuit.

3 The **workbench** which is where almost all interaction with GatePlay happens. When editing you can move, delete, and draw wires between

[1] http://logic.ly
[2] http://www.circuitlab.com

- **Input** components such as constant ON, $Toggle$ (clicking on a $Toggle$ during simulation will chance its value), and $Blinker$ (which flip their value at a constant interval)
- **Gates** are some standard boolean logic functions such as NOT, AND, and XOR
- **Composites** like half and full adders
- **Memory** components like SR latches and D flip-flop

Figure 1.1: Overview of the components available in GatePlay

components. When simulating the values of wires are shown visually on the workbench.

GatePlay has a library of standard components to build circuits with:

Figure 1.3 shows a D flip-flop being simulated on the workbench. Green wires are $High$ (logical $True$) and red wires are Low (logical $False$). The round components on the left are $Toggle$s; clicking them will change their value and cause the resulting changes to propagate through the circuit.

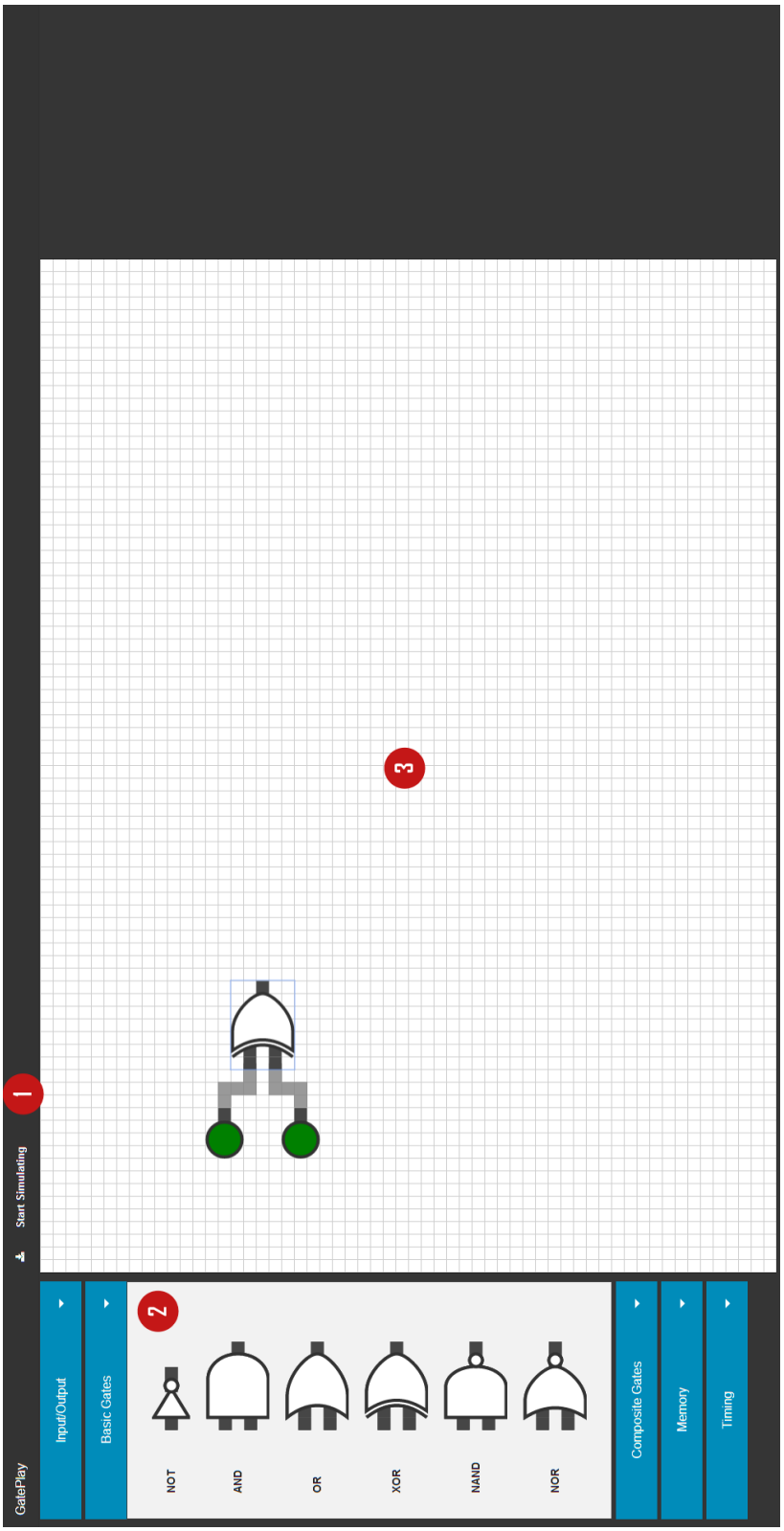


Figure 1.2: Drawing a circuit

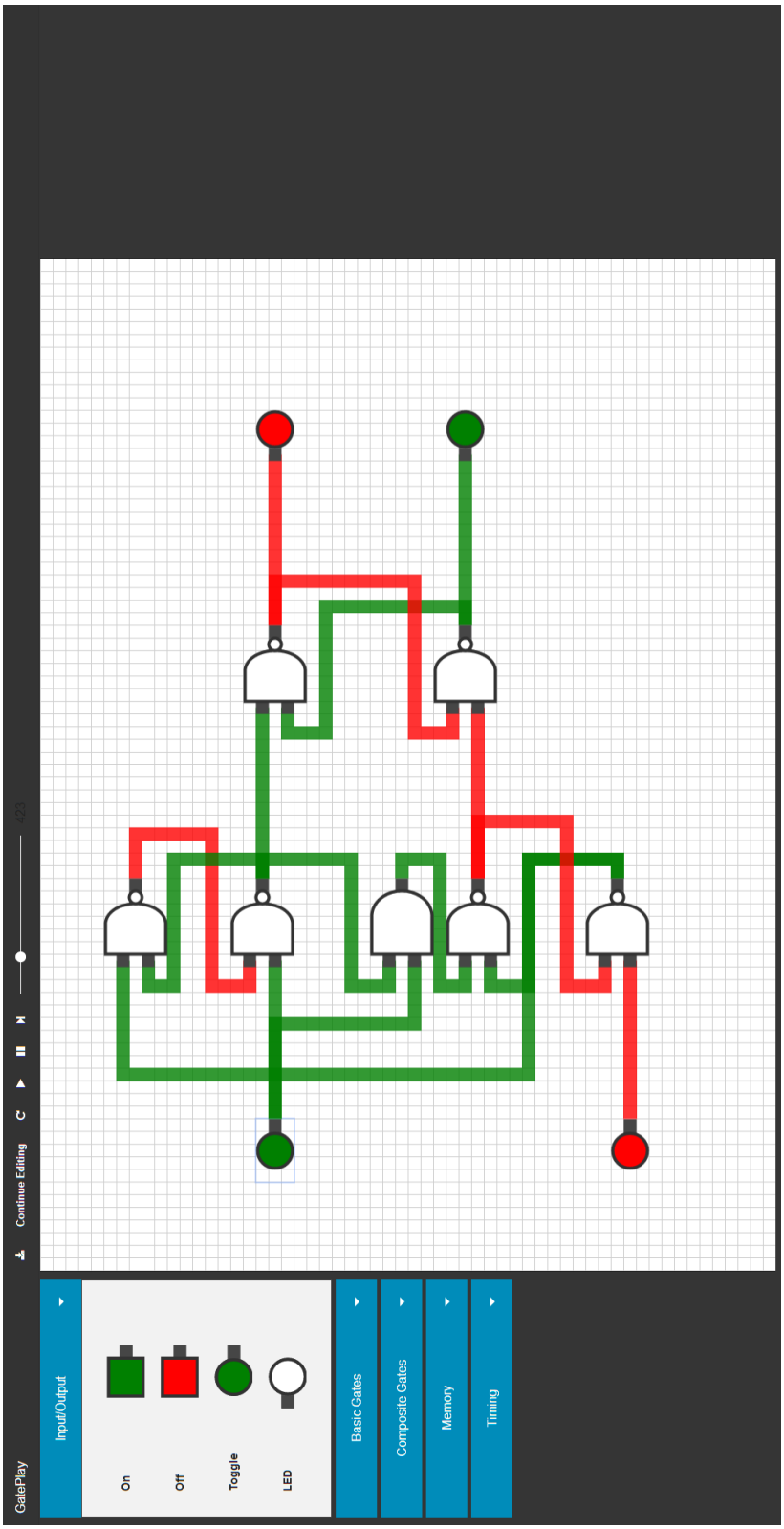


Figure 1.3: Simulating a D flip-flop

Chapter 2

Background

2.1 Logic Circuits

A logic circuit is a collection of logic *gates* to perform logical operations on data. They are used to implement Boolean functions, which in turn can run all algorithms which can be described with Boolean logic.

2.2 Building Websites

GatePlay is a web application. Instead of being installed on the user's computer it runs in their web browser. GatePlay is a single HTML document and has relatively few CSS style rules. The vast majority of the work was creating the JavaScript, which handles all interactions with the top and left bars, as well as displaying and editing circuits.

HTML defines the semantic structure of the website as a collection of nested elements, known as the DOM (Document Object Model) tree. In other words, HTML defines the content of a website and the structure of headings, sections, paragraphs, etc.

CSS Cascading Style Sheets modify how HTML documents are displayed by the clients browser. CSS files are lists of *selectors* with associated *attributes*. A selector describes elements based on their position in the DOM tree, and its attributes modify how those elements are displayed. For example, it is easy to specify the following styles in CSS: "All elements of type *gate* should be 150 pixels wide", or "The middle section of the website should take up 80 percent of the width".

JavaScript JavaScript is a full programming language which is run in the user's browser when they load the website. It can perform arbitrary computations, and is also free to modify the DOM tree and the styles of elements.

Chapter 3

Requirements

The core goal of the project is to provide a workbench for logic circuits which is as accessible to as many people as possible, without being over-simplified.

Chapter 4

Simulation

4.1 Modelling Circuits

A circuit is a set of components and a set of wires. The fields which define components and wires are shown in figures 4.1 and 4.1 respectively. The only constraint placed on circuits is that no more than one wire may go into the same input port.

- The **number of inputs** the component has (N)
- The **number of outputs** the component has (M)
- An **evaluation function** which takes a list of N truth values and returns a list of M truth values. The Evaluation Function also defines the propagation delay of the component, as discussed throughout this chapter.

Figure 4.1: Fields of a Component

- The **source component** the wire is leaving from
- The **output port** of the source component
- The **target component** the wire is going to
- The **input port** of the target component
- The current **truth value** of the wire

Figure 4.2: Fields of a Wire

4.2 Event-Based Simulation

GatePlay uses an event-based algorithm to simulate logic circuits. An event is a notification that a specific output of a component has changed. They are defined by the following four values:

- The **source component** the event is propagating from
- The **output port** on the source component which has changed value

- The **event time** at which it is occurring
- The new **truth value** of the output port

Events are stored in a priority queue, and have priority equal to their event time. Lower times are more urgent.

The heart of an event-based simulation is the event loop which processes all the events and generates new events.

1. **Fetch** next event to be processed from priority queue
2. **Update** the value of the wires connected to the event port
3. **Recalculate** the output of any gates whose inputs changed
4. **Propagate** the change by creating new events for each changed output, and add to the queue after the gate's delay

Figure 4.3: Event Loop Body

4.2.1 Event Loop Example

Consider the circuit shown in figure 4.4. Let X's current output be $False$, the event queue contain a single event $ev = (X, 1, t, True)$, and the current time be t.

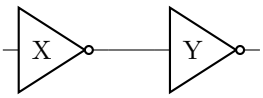


Figure 4.4: Two NOT Gates

When the event look body executes, the following actions occur:

1. Pop ev from the queue
2. Update the wire coming from X port 1 to $True$
3. A enters Y, so its input has changed. Apply Y's evaluation function to the new input, which returns $False$
4. Since Y's output changed, add the following new event to the queue: $(Y, 1, t + \delta, False)$ where δ is Y's gate delay

As the event loop repeats, changes propagate through a circuit.

4.2.2 Culling

Events can sometimes be discarded without being processed by the entire event loop. For example, events which set an output port to the value it already is do not change anything in the circuit and can be discarded during stage 2 of the event loop.

4.2.3 Initial Components

We have seen events being generated as a result of previous events. However some components, known as *Initial Components* can create events spontaneously.

- **Constant Components** such as ON and OFF never change value, and so place their events in the queue just once when the circuit is initialised.
- **Timed Components** such as *Blinker*s toggle their output at a set interval. *Blinker*s must be periodically polled to see if they have changed value. In GatePlay's implementation all blinkers add an event on every clock tick and we rely on culling (see 4.2.2) to eliminate tautological events. A more efficient implementation would be to use a priority queue to store the next time each blinker is going to change value. The naive implementation gives acceptable performance when there are relatively few *Blinker*s in the circuit. If users were found to be using too many *Blinker*s it would be possible to implement the better algorithm.
- **External Components** like *Toggle*s add events based on user interaction. Since the stimulus to create an event comes from outside the simulator, it is a simple case of putting a method to create circuit events in the circuit's public API.

4.2.4 Event Race Conditions

Consider the circuit shown in figure 4.5:

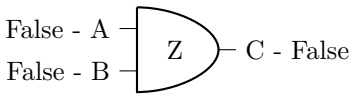


Figure 4.5: AND gate

Suppose there are two events in the event queue: $(X, 1, t, True)$ and $(Y, 1, t, True)$ where X port 1 is wired to A, and Y port 1 is wired to B. In other words, A and B become $True$ at time t. It now becomes time t.

We handle each event independently. Suppose that we handle the event coming from X first. We set A to $True$, B is still thought to be $False$, and therefore we add the event $(Z, 1, t + \delta, False)$ to the queue (as $TrueANDFalse = False$).

Next we handle the event coming from Y. B is set to $True$ and A is known to be $True$, so we add the event $(Z, 1, t+\delta, True)$ to the queue ($TrueANDTrue = False$).

The queue now contains two events with different truth values occurring on Z port 1 at the same time. If the event valued at $False$ is handled first it will be culled and the circuit will be simulated correctly. However if it is handled second then the output of Z will be calculated as $False$, despite both its inputs

being $True$! Since we are using a priority queue and both events have the same priority, there is no defined behaviour for which event will be handled first.

The solution is to do a first pass through all events happening at time t and update the values of each of the affected wires. Following that with the previous algorithm will result in both generated events being $(Z, 1, t+\delta, True)$ and there will not be a race condition.

A further refinement implemented in GatePlay is to use a set to store the components whose inputs have been changed. Since sets do not store duplicates, Z's output will only be calculated once.

4.2.5 Efficiency

The speed of an event-based simulation is proportional to the number of events being generated and processed. For this reason, reducing the time spend processing events through Culling (see section 4.2.2) is critically important.

Also note that (if sensible data structures are used) event-based simulations are still performant in circuits with very large number of components and connections, so long as relatively few events are occurring.

4.3 2-Valued Simulation

4.3.1 Description

A simple simulation of logic circuits might use a 2-valued simulation. Each wire has one of two values: *True* or *False*. While it provides a reasonable model for logic circuits, it has some limitations described in this section.

4.3.2 Initialisation Values

In a 2-valued simulation all wires must be either $True$ or $False$ valued. It is necessary to initialise the wire values before the circuit begins, but what is a sensible default? Suppose we initialise all wires to $False$, and consider the circuit in figure 4.6:

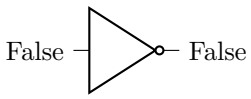


Figure 4.6: An inconsistently initialised NOT Gate

To have both A and B both be initialised to be $False$ is inconsistent with the logic of the circuit. The same would be true is wires were initialised to $True$. One option would be to add a boolean flag to wires indicating that no event has reach them yet, and leave them uninitialised at first, however there is a more suitable solution introduced in section 4.4.

4.3.3 Propagation Delay Uncertainty

As discussed in section 2.1, the *Propagation Delay* of a component is the time it takes from its inputs being stable and valid to its outputs becoming stable and valid (from Wikipedia).

Previously we have assumed that this delay is constant for a given component, but in reality the precise delay varies based on temperature, voltage, and output capacitance. Our model of logic circuits does not consider these factors and therefore cannot make an informed estimation of the delay for each pass through a component.

For example, suppose the *NOT* gate shown figure 4.7 never has a propagation delay of less than δ_{min} nor a delay of greater than δ_{max}.

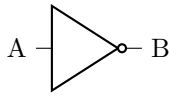


Figure 4.7: A *NOT* Gate

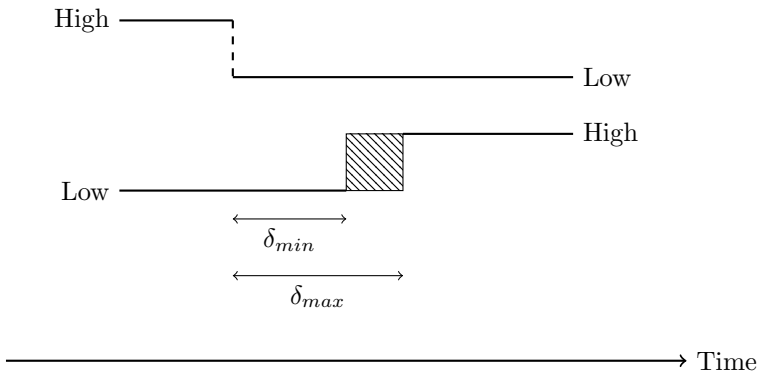


Figure 4.8: 2-valued Low-High transition

If we flip A's value from $True$ top $False$ the trace of the truth values through time is shown in figure 4.8. B's transition from Low to High can happen at any point in hatched region based on the aforementioned factors. A 2-valued simulator at this level of detail can therefore only assume that the transition happens at a random time. In other words, the precise propagation delay for each pass through a component is randomly sampled from interval $[\delta_{min}, \delta_{max}]$.

This is not desirable behaviour for a simulator, as each simulation of a circuit will likely have different propagation delays which can potentially change the output of a circuit altogether. If a run of this simulation works as desired on a circuit there is no guarantee that *all* runs would yield the desired result.

4.4 3-Valued Simulation

The way I decided to overcome the problems of 2-valued simulation (see section 4.3) was to instead use a 3-valued simulation. A wire has one of three values: $True$, $False$, $Unknown$. A wire that is $Unknown$ may "in reality" be either true or false, but the simulation does not know which.

4.4.1 Initialisation Values

In a 3-valued simulation all wires can be initialised to $Unknown$ without the problem of inconsistency for 2-valued simulations discussed in section 4.3.2.

4.4.2 Propagation Delay Uncertainty

The 2-valued simulation was unable to model the uncertainty of component delay without making the result of each simulation run non-deterministic as discussed in section 4.3.3.

Suppose we have a component X whose inputs are changing at time t. In the 2-valued simulation we would consider the outputs changing at time $t+\delta$ where δ is randomly sampled from the interval $[\delta_{min}, \delta_{max}]$.

However in the 3-valued simulation we can consider the outputs of X to be changing twice. At time δ_{min} they becoming $Unknown$, and then at time δ_{max} they settle on a stable value.

By considering the output $Unknown$ for the duration of the delay uncertainty we have still modelled the uncertainty in the actual components, but without introducing uncertainty to our simulation. In other words, should a circuit yield a certain value in our simulation we know that it will yield a certain value for *all* possible combinations of gate delays.

4.4.3 Event Suppression

A bug was introduced in the simulator in section 4.4.2. Consider the circuit shown in figure 4.9. Let δ_{min} and δ_{max} be the minimum and maximum gate delay respectively. A, B, and C are all $False$.

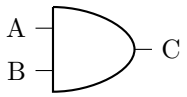


Figure 4.9: An AND Gate

Let circuit event e_1 set A to $True$ at time t, and event e_2 set B to $True$ at time $t+1$. Using the algorithm thus far described we will handle e_1 and the following two events will be added to the queue: $e_3 = (X, 1, t+\delta_{min}, Unknown)$,

$e_4 = (X, 1, t + \delta_{max}, False)$. Next, handling $e2$ we will add these events to the queue: $e_5 = (X, 1, t + 1 + \delta_{min}, Unknown)$, $e_6 = (X, 1, t + 1 + \delta_{max}, True)$.

The bug arises at time $t + \delta_{max}$ when we handle e_4. At that time C is set to $Unknown$ and we will set it to $False$. However this is incorrect behaviour as e_2 means that C should be uncertain until e_6 is handled at time $t + 1 + \delta_{max}$.

In other words, the output of X is uncertain for two distinct reasons. e_1 causes C to be uncertain in the interval $[t + \delta_{min}, t + \delta_{max})$, and e_2 makes it uncertain in the interval $[t + 1 + \delta_{min}, t + 1 + \delta_{max})$.

The simulator therefore must keep track of the intervals a wire is uncertain for, and *suppress* any events which try to set it to a certain value in any of those intervals. In this example e_4 must be suppressed.

4.4.4 Performance Considerations

When implementing a 3-valued event-based simulation it is critically important to consider carefully the conditions under which events are *culled*. Many events will cause two more events to be created (an $Unknown$ event followed by either a $True$ or $False$ event). To avoid an exponential explosion in the number of events being processed, duplicate events or events which do not change values in the circuit must be discarded.

Chapter 5

Implementation

Make some general points here

5.1 Getting Started

5.1.1 Require.js

When the GatePlay url is visited, the only file downloaded is the main HTML file: index.html. The index then directs the user's browser to download the additional CSS and JavaScript files needed to use GatePlay.

A JavaScript script may used methods or variables defined in the global namespace, even if they were put there by another script. The only restriction is that

```
1 <!-- componentview depends on component -->
2 <!-- Therefore we ensure component is loaded first -->
3 <script type="text/javascript" src=".../component.js"></script>
4 <script type="text/javascript" src=".../componentview.js"></script>
```

Figure 5.1: Example use of Script tags

It is time consuming for a human to find and type out a correct ordering for the Script tags, and it would need to be updated every time a file as added or removed, or sometimes if a file were modified.

Require.js is a JavaScript file loader which does automatically loads files in a correct order. Each JavaScript file declares each of its direct dependencies, and Require.js will ensure they are all loaded correctly when the webpage loads.

```
1  // componentview.js
2
3  require([
4    // Declare the path of each file we require
5    "canvas/model/component"
6  ], function(Component) {
7    // Each included file is run, and we can give a name to whatever
         it returns if desired
8    var myComponent = new Component();
9    ...
10 });
```

Figure 5.2: An example file which uses Require.js

5.2 The Workbench

GatePlay's workbench is where we create and view circuits. It is implemented using an HTML Canvas element. It is a blank slate which can have shapes and images drawn using the JavaScript API.

5.2.1 Fabric.js

An HTML Canvas only provides low level drawing tools. You are able to draw shapes and images on it, but there is no concept of persist objects on the canvas.

Fabric.js is a library which wraps HTML Canvases with an object model, allowing GatePlay to interact at the level of objects being added to, modified, and remove from the canvas.

Suppose I wanted to add a rectangle to a canvas, and then move it to a new location. Using Fabric.js this is two library calls (one to add a rectangle object and one to change the position property of the object). Using an HTML Canvas it is still one call to draw the rectangle, but moving it would require calculating what would be behind the rectangle, drawing that over the rectangle, and then re-drawing the rectangle at its new location.

Initially GatePlay used a different canvas framework called KineticJS, but due to difficulties getting features like snap-to-grid working I switched to Fabric.js.

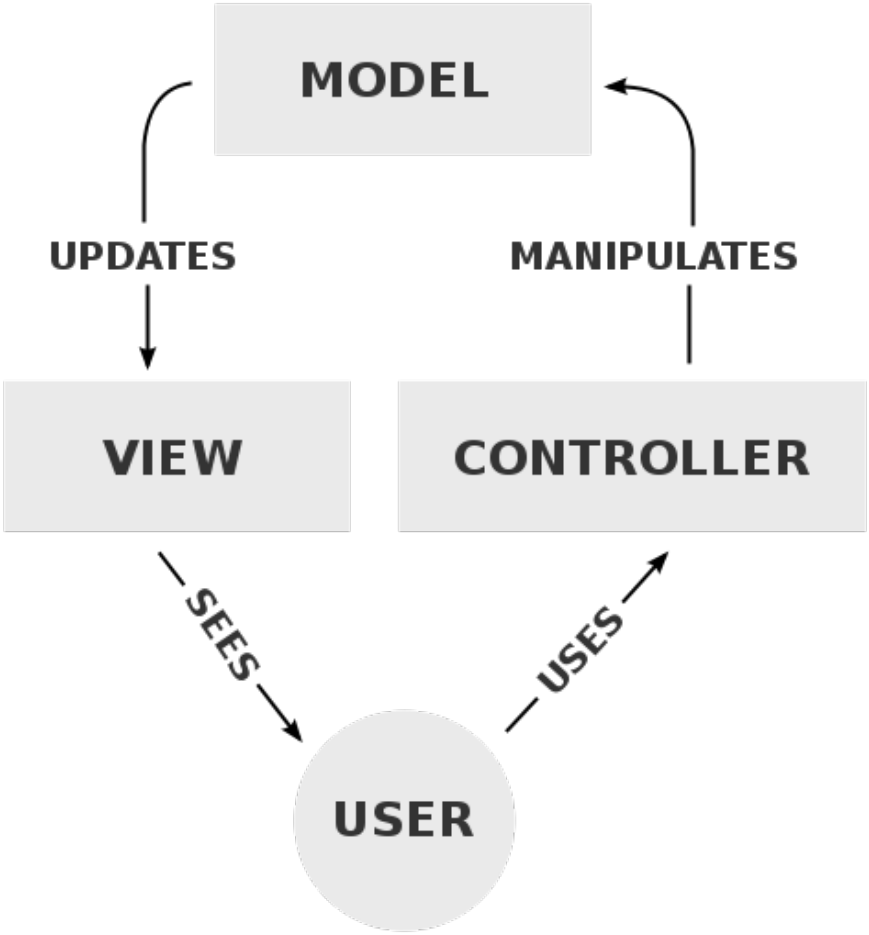
5.2.2 MVC with Backbone.js

Model-view-controller[1] is a design pattern to simply program development. GatePlay uses MVC in the implementation of the workbench. MVC applications contain three types of components:

- A **model** which stores some part of the state of the application
- A **view** which displays a representation of one of the models to the user

[1] http://en.wikipedia.org/wiki/Model-view-controller

* 0.5*



0.5

Figure 5.3: Interaction of MVC Components, from Wikipedia

- A **controller** which processes user input, and updates the appropriate models.

Backbone.js[2] is a JavaScript library to reduce the amount of boilerplate code in developing MVC JavaScript applications.

One model is the Wire model as shown in figure 5.2.2. It includes a lot of the same information as the Wire objects used in the simulation, but also includes information regarding the *fixed points* of the wire.

We then create a view associated with the Wire model (figure 5.2.2).

[2]http://backbonejs.org

```
Backbone.Model.extend({
  defaults: function() {
      return {
          id: nextWireId++,
          sourceId: -1,
          sourcePort: -1,
          targetId: -1,
          targetPort: -1,
          fixedPoints: [],
          truthValue: TruthValue.UNKNOWN
      }
  },
})
```

captionDefinition of a Wire model

```
Backbone.View.extend({
    initialize: function(options) {
        // When the model is changed, update the view
        this.model.on("change:fixedPoints", this.render, this);
        this.model.on("change:truthValue", this._setWireColor, this
            );
    },

    render : function() {
        var model = this.model;

    // Using the model data we can now draw wires on the canvas
    },

    _setWireColor: function() {
        var truthValue = this.model.get("truthValue");

        // We can now re-render the wire with the new colour
    }
});
```

captionDefinition of a Wire view

5.2.3 Editing Mode

5.2.4 Running Mode

5.3 The Simulator

The implementation of GatePlay's simulator is simply a transliteration of the algorithms and ideas explained in section **??** to JavaScript. The following six classes fully implement the simulator:

- **truthvalue.js** defines constants $TRUE$, $FALSE$, and $UNKNOWN$
- **component.js** defines a Component by its input count, output count, and evaluation function
- **wire.js** defines a Wire by its input component and port, output component and port, and truth value
- **circuitevent.js** defines a CircuitEvent by component, port, timestamp, and value
- **functions.js** contains definitions of all the Evaluation Functions available to the simulator
- **circuit.js** is the only class which need be visible from outside the simulator. It has an interface to add components and wires. Circuit.js implements the algorithm for the event loop.

5.4 Drag and Drop

One of the requirements of GatePlay was that it be easy to use, and I felt the most natural way to add components to the circuit is to drag them on from the side.

5.5 Tying GatePlay Together

5.5.1 application.js

Chapter 6

Testing

Testing is extremely important in the development of non-trivial programs. Tests give some assurance to the correctness of the code, and highlight *regressions* (where code that used to work is broken by a recent change) quickly.

6.1 Unit Testing

Unit testing is used to test the correctness of small modules of code, such as functions. The simulator has a suite of unit tests. One example is a test over the AND evaluation function shown in figure 6.1. All *evaluate* functions had to be changed when *Blinker*s were implemented, and this test flagged a regression when the first implementation had a bug.

Unit tests are also very useful for catching simple bugs in frequently edited code. For example, the *tick* function in the circuit simulator is responsible for handling all events which occur at the current time, and them increments the current time. Since the *tick* function is so frequently modified, it is easy to introduce simple bugs. The test shown in figure simply checks that the system clock is incremented after *tick* is called.

I used a common JavaScript unit testing framework QUnit[1] to reduce the boilerplate in writing unit tests.

6.2 End-to-end Testing

Unit testing is a bottom-up approach which checks that the smallest modules work as expected in isolation. End-to-end testing is a top-down approach which

[1] http://qunit.com

```
var T = TruthValue.TRUE;
var F = TruthValue.FALSE;
var U = TruthValue.UNKNOWN;

function ANDTest() {
  var and = Functions.get("and");

  // Test 2-input truth table
  equal(and.evaluate([T,T]), T);
  equal(and.evaluate([T,F]), F);
  equal(and.evaluate([T,U]), U);
  equal(and.evaluate([F,T]), F);
  equal(and.evaluate([F,F]), F);
  equal(and.evaluate([F,U]), F);
  equal(and.evaluate([U,T]), U);
  equal(and.evaluate([U,F]), F);
  equal(and.evaluate([U,U]), U);
}
```

Figure 6.1: Testing AND's truth table

Chapter 7

Conclusions

Bibliography

[1] http://logic.ly.