Logic Circuit Workbench: GatePlay

Greg O'Connor
Computer Science, University of Oxford

May 16, 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. It is therefore important that as many people as possible have the opportunity to learn about and understand logic circuits.

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.

I wanted to build a logic circuit workbench which was very simple to access and use, but without simplifying the simulation to the point where it is unhelpful. As modern browsers go to great lengths sandbox a website's JavaScript programs from the user's system, there is no need for the users to trust my implementation to use it.

1.2 Overview of GatePlay

The main interface of GatePlay can be seen in figure 1.1. The labelled regions are:

1 The **top bar** 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** 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** is where almost all interaction with GatePlay happens. When editing you can move, delete, and draw wires between components. When simulating the values of wires are shown visually on the workbench.

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

Figure 1.2 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.

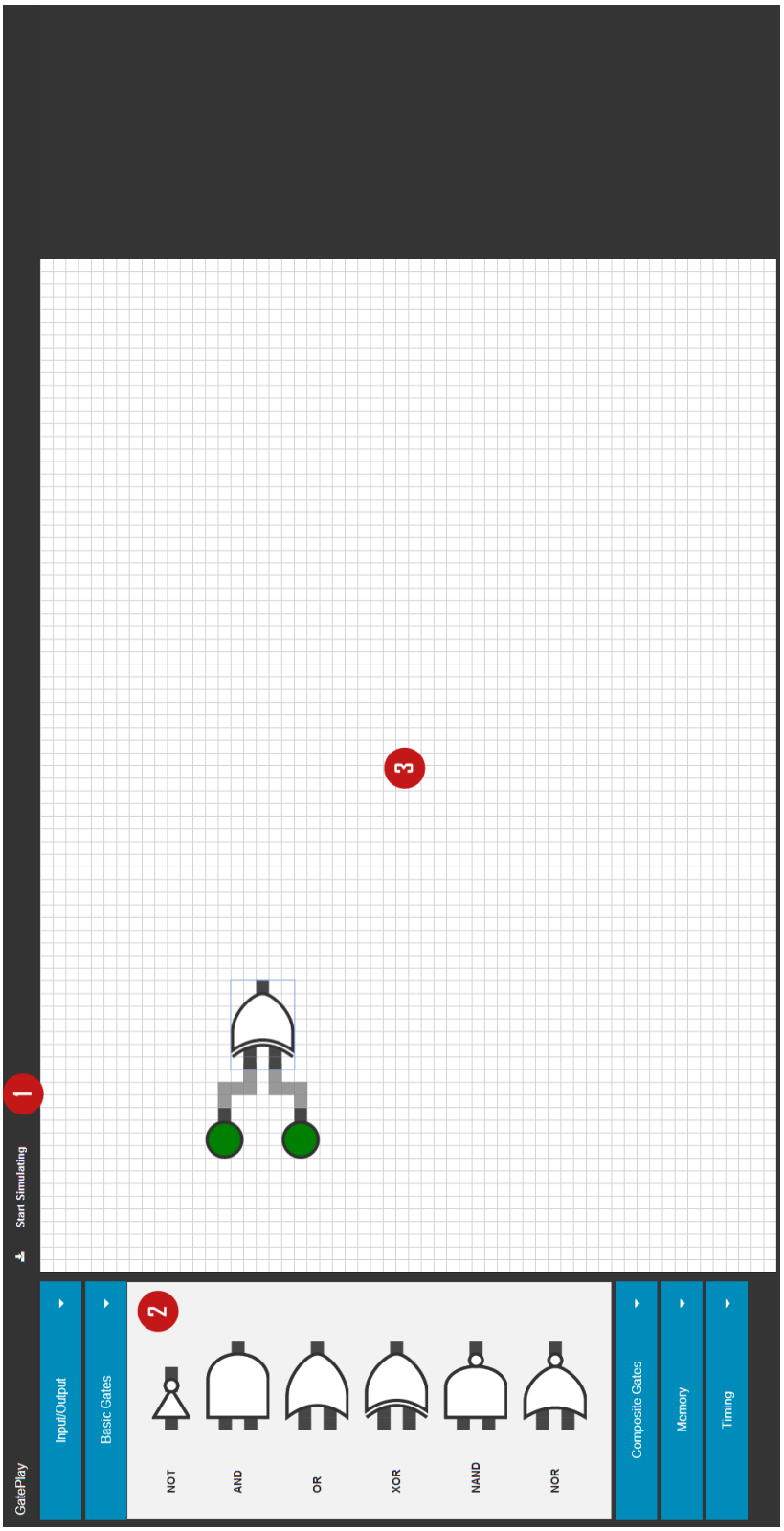

Figure 1.1: Drawing a circuit

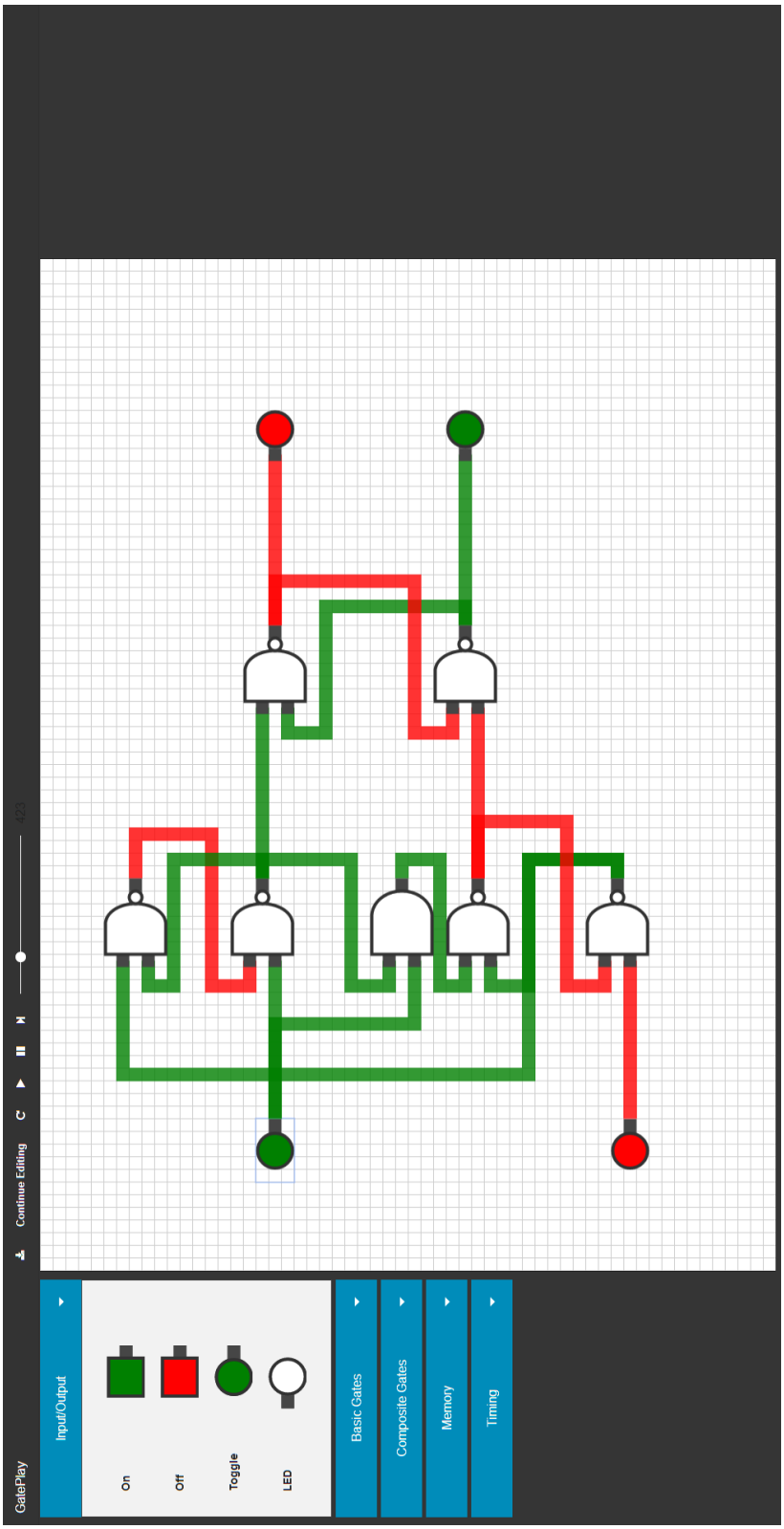

Figure 1.2: Simulating a D flip-flop

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, which is loaded into the clients browser as a web page. Practically all websites use HTML and CSS, and possibly JavaScript, for their implementation.

2.2.1 HTML

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.

2.2.2 CSS

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".

2.2.3 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 be incident on a single 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 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 A's output current output be $False$, event queue contain a single event $ev = (A, 1, t, True)$, and the current time be t.

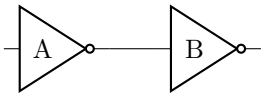

Figure 4.4: Two NOT Gates

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

4.2.2 Culling

Events can sometimes be discarded without being processed by the entire event loop. For example, events which set a wire 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

So far we have only discussed 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 once when the circuit is initialised.
- **Timed Components** such as *Blinker*s toggle their output at a set interval and cannot be fully handled when the circuit is initialised. *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 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 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:

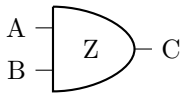

Figure 4.5: And gate

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. Before handling the events, A, B, and C are all $False$. It now time t.

We handle each event independently. Suppose without loss of generality 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.

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.

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 largely determined by the number of events being generated and processed, provided sensible data structures are used to store the components and wires.

Event-based simulations are therefore particularly performant in quiet circuits, where there may be a large number of components and connections but fewer changes are occurring.

Reducing the time spend processing events through Culling (see section 4.2.2) is critically important.

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

Another problem with a 2-valued simulation is the matter of initialising a circuit before simulation begins. All wires must be either $True$ or $False$ valued, but what is a sensible default? Suppose we initialise all wires to $False$. Consider the circuit in figure 4.6:

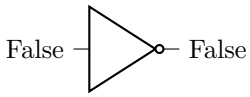

Figure 4.6: An inconsistently initialised NOT Gate

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 were it initialised with all $True$ values. It is possible to solve the problem by moving to a 3-valued simulation (discussed 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}.

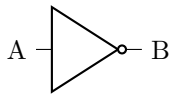

Figure 4.7: A NOT Gate

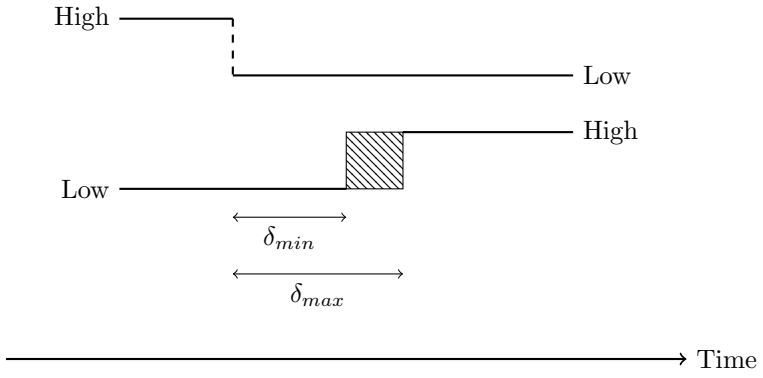

Figure 4.8: 2-valued Low-High transition

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. In other words, 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 $Unknown$ is "in reality" either true or false, but it 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 $False$.

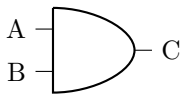

Figure 4.9: An AND Gate

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

5.0.5 Loading Javascript and Require.js

When the GatePlay site 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 run GatePlay.

JavaScript files are imported into a webpage using the Script tag. For a file to be correctly loaded all of its dependencies must have been run already.

However it is time consuming for a human to find and type out a correct ordering for the imports, and would potentially need to be updated every time a file as added or modified.

Require.js is a JavaScript file loader which does this automatically. 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 });
```

```
1  ...
2  <!-- componentview depends on component, so we ensure component is
       loaded first -->
3  <script type="text/javascript" src=".../component.js"></script>
4  <script type="text/javascript" src=".../componentview.js"></script>
5  ...
```

Figure 5.1:

5.0.6 jQuery

Different browsers provide subtly different ways for JavaScript to interact with the DOM tree - making multi-browser compatible code tedious to write and maintain.

jQuery is a ubiquitous JavaScript library which provides a single API for common JavaScript actions across many versions of different browsers:

```
1  require([
2    "lib/jquery".
3  ], function($) {
4    // Select elements which are of class "gate"
5    var gateElements = $(".gate");
6
7    // Set the width of each gate to 100px
8    $(gateElements).each(function() {
9      $(this).width("100px");
10   });
11 });
```

5.1 Canvas

GatePlay's main canvas is where we view and edit circuits. It is an HTML Canvas element, which

5.1.1 Fabric.js

HTML Canvas elements provide only low level drawing tools. You are able to draw shapes and images on them, 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 is 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 at the time the API documentation was not as good and so I transitioned to using Fabric.js.

5.2 Simulator

GatePlay's simulation is a plain JavaScript implementation of the concepts explained in

5.2.1 TruthValue

5.2.2 Component

A component is defined by the following class constructor:

```
function Component(id, funcId, inputCount, outputCount, cArg) {
    this._id = id;
    this._funcId = funcId;
    this._inputCount = inputCount;
    this._outputCount = outputCount;
    this._cArg = cArg;

    this._evalFunc = Functions.get(funcId, cArg);
}
```

id
: Is the unique identifier of this instance of a component. It is provided as a parameter in the constructor because it is simpler to use the same id for the object drawn on the canvas and the Component in the simulation.

funcId
: This is the id of the evaluation function of the Component. For example an AND gate would have funcId being "and".

inputCount

outputCount

cArg
: An optional parameter for gates which require it. For "blinker" components cArg is set to the period of the blinker. For simpler components like "and" it is unset.

evalFunc
: We fetch the appropriate evaluation function from the function store

5.2.3 Evaluation Functions

An evaluation function is first defined by the abstract class EvaluationFunction. It provides code common to all Evaluation Functions, such as checking that the list provided to the function "evaluate" is of a suitable size.

```
// Define the class with two parameters
function EvaluationFunction(minInputCount, maxInputCount) {
    this._minInputCount = minInputCount;
    this._maxInputCount = maxInputCount;
}

// Define "evaluate" function
EvaluationFunction.prototype.evaluate = function(argList, clock) {
    if (typeof argList == "undefined" || !argList instanceof Array)
        throw "argList must be a list";

    if (argList.length < this._minInputCount || argList.length >
        this._maxInputCount)
        throw "Invalid number of arguments passed to
            EvaluationFunction";
```

```
14
15      return this._doEvaluate(argList, clock);
16  };
17
18  // "_doEvaluate" must be overridden by subclasses of
        EvaluationFunction
19  EvaluationFunction.prototype._doEvaluate = function(argList, clock)
        {
20      throw "_doEvaluate not implemented on EvaluationFunction, you
            must override it";
21  }
22
23  // Default gate delay is 5 clock cycles
24  EvaluationFunction.prototype.getDelay = function() {
25      return 5;
26  }
27
28  // Default uncertainty duration is 1 clock cycle
29  EvaluationFunction.prototype.getUncertaintyDuration = function() {
30      return 1;
31  }
```

Individual implementations of evaluation functions then inherit from EvaluationFunction.

```
1    // Define the "And" class
2      function And() {
3      }
4
5      // Inherit from EvaluationFunction
6      // We define that an And gate must have at least two inputs,
           but there is no upper bound (we can simulate n-input AND
           gates)
7      And.prototype = new EvaluationFunction(2);
8
9      // Provide an implementation for "_doEvaluate"
10     And.prototype._doEvaluate = function(argList, clock) {
11         var onlyTrue = true;
12
13         for (var i = 0; i < argList.length; i++) {
14             var v = argList[i];
15
16             // If any of the arguments are FALSE, return FALSE
17             if (v === TruthValue.FALSE) {
18                 return [TruthValue.FALSE];
19             }
20             onlyTrue = onlyTrue && v === TruthValue.TRUE;
21         }
22
23     // If all values are TRUE, return TRUE
24         if (onlyTrue) {
25             return [TruthValue.TRUE];
26
27     // Otherwise we have zero FALSE values and at least one UNKNOWN
            value, so the output is UNKNOWN
28         } else {
29             return [TruthValue.UNKNOWN];
30         }
31     };
```

Because our implementation of the And function does not override the "getDelay" and "getUncertaintyDuration" functions, it has the default values of 5 and 1 respectively.

5.2.4 Wire

Wires are defined as follows for the simulation

```
define([
    "sim/truthvalue"
], function(TruthValue) {
    function Wire(id, sourceId, sourcePort, destId, destPort) {
    // Wire id
        this.id = id;

        // Id of source component
        this.sourceId = sourceId;

        // Output index from the source component
        this.sourcePort = sourcePort;

        // Id of the destination component
        this.destId = destId;

        // Input index to the destination component
        this.destPort = destPort;

        // Current truth value of the wire (starts as UNKNOWN)
        this.truthValue = TruthValue.UNKNOWN;

        this.unstableUntil = 0;
    }

    return Wire;
});
```

5.2.5 application.js

Chapter 6

Testing

Chapter 7

Conclusions

Bibliography

[1] http://logic.ly.