Sifteo SPICE

by

Ikenga Kenneth Ugo

Senior Project

COMPUTER ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

JUNE 2013

TABLE OF CONTENTS

Contents

I. Acknowledgements

I would like to thank...

Dr. Oliver for being my advisor and allowing me to take on the experience of Sifteo game development. I really appreciate all the support through all the hardships I had to face while working on this project.

Anjelica Concepcion for being my Sifteo partner this year and helping me discover and fix issues that I may not have been able to otherwise. All the extra experience was essential to the completion of this project.

Karina Cordon and Dereck Quock for setting an example with your previous work in Sifteo development. I would also like to thank Sean Voisen for a very valuable tutorial that provided me with the tools to get started and a reference for potential issues that I may not have been able to fix otherwise.

Lastly, I want to thank Kevin Peters for his essential work on a similar project and all the insights he provided.

II. Abstract

Sifteo SPICE is an application built on the Sifteo platform with the purpose of aiding in the learning of basic circuits. Traditionally, circuit theory is taught to students in two different methods, lectures and laboratory exercises. Lectures focus on auditory and visual learning and are largely passive learning. Lab experiments allow students to physically interact with the circuits, and learn visually through viewing output waveforms from simulators or on measurement devices.[3] The goal of the Sifteo SPICE project is to develop a physical system for virtual, real-time SPICE simulation that mimics the laboratory experience. In Sifteo SPICE, each individual Sifteo cube acts as a circuit node that communicates with immediate neighbors using near-field communication. Each cube also utilizes various sensors in order to specify each circuit component and component value. Additionally, the cubes communicate wirelessly with a host computer, running a customized version of SPICE. When the user chooses to run the analysis on the circuit, data is aggregated on the host computer and plotted in real-time.

III. Introduction

Every Electrical and Computer Engineering major coming through California Polytechnic State University, San Luis Obispo (Cal Poly) must learn the basics of circuit analysis. The primary method of learning is during lectures, which appeals to the auditory and visual learners, but many students still require the hands-on experience and immediate feedback demonstrated during the laboratory exercises to truly comprehend what is actually taking place in circuitry. Currently, these labs rely on two techniques – SPICE simulations and breadboard analysis using measurement equipment. Each of these methods has advantages and disadvantages in learning the circuit concepts. The method of breadboarding a circuit requires a lot more time and equipment. To change a resistance, the part must be replaced, or a decade box has to be used. After you have the desired components, each part must be wired together, and more wires used for the measurement equipment. If something is configured incorrectly, including incorrect wiring, wrong values, or faulty components, the measurements will be incorrect, or even worse, a component or piece of equipment may blow out or break.[3]

SPICE provides a virtual environment for students to place any component at any value in any configuration without the risk of breaking any components. These simulations allow the users to plot out information relating to the voltage or current at any point in the circuit. The drawbacks of using SPICE to simulate circuits include a small learning curve (especially when using the netlist entry method), no option to simulate in real-time, and a severe disconnect from building a circuit. Through the previous experience comparing traditional circuit modeling and using SPICE, it is clear that the use of SPICE increases the ability for students to learn concepts. At the same time, there are some limitations to the ease of use for students relating

to the specifications of simulation parameters. When SPICE is applied to educational purposes, exercises are adjusted to accommodate for SPICE. To improve its effectiveness, a solution should be developed that evolves SPICE into a simple, user friendly tool with fast results. [3]

a. Solution

In order to improve the current methods for simulating circuits in lab, a new simulation tool was developed that includes the feel of building a circuit with the virtual nature of SPICE, while reducing the knowledge of SPICE and simulation parameters requirements. The solution involves using Sifteo cubes to select components and build a circuit, which will then be simulated on a host computer running SPICE and updating information whenever the user chooses. This method for simulating circuits combines the efficiency and speed of SPICE with the physical feel of building a circuit creating a physical-virtual interface. Simulating a circuit requires the physical interaction for selecting components and wiring a circuit together, but uses virtual models for the simulation. By creating this unique solution called Sifteo SPICE, the amount of SPICE knowledge required has been reduced, and using a simple interface such as Sifteo cubes, the ability of a student to learn the concepts with ease should be increased. Utilizing Sifteo SPICE to assist students in the learning of electrical engineering circuit concepts should be a valid and valuable asset to anyone who may need extra help regarding circuit comprehension.[3]

b. Sifteo SPICE System Overview

The Sifteo SPICE simulator is built around Sifteo cubes and coordinated by a host computer. Each Sifteo cube device will represent a component selected by the user. The currently supported components include a resistor, capacitor, inductor, and DC voltage source. All of these component screens, as well as the component selection screen can be seen in Appendix C. After the components have been selected on the cubes, the inputs and outputs of these circuit components can be chosen by touching either the top or bottom of the cubes together. When a component has been established in the circuit – the input and output of the circuit component have been defined – the user will be able to select a corresponding value. Once the circuit is built, the host computer's functionality can be initiated by the user.

The host computer's role begins with taking the configuration of the circuit and creating the corresponding netlist. The computer checks each cube to see whether it has a defined input, output and component value. If the cube passes this check, it is added to the netlist. After all the cubes have been checked and the netlist is deemed complete a SPICE simulation is run using the netlist. The resulting output of this analysis is a text file with a table containing the analysis specified by the netlist. Then this table is run through a plotting program to create a graph visualizing the analysis. Any updates to the circuit are possible but require another initiation by the user in order to display the new analysis.

IV. Background

The original Sifteo cubes were an interactive gaming platform released in September 2011. Created by Sifteo, Inc., these blocks have a length and width of 1.5 inches and a height of about .5 inches. They each have a clickable full-color LCD screen, a 3-axis accelerometer, and near-field communication technology between the cubes. Sifteo cubes were sold in sets of 3 or 6 cubes and came with a charging dock, which also served as a protective carrying case. To use the original cubes, it was necessary to have either a Mac or PC, as well as the ability to download the Sifteo software. Games could be purchased and downloaded through the online Sifteo store and then installed onto cubes using a USB connector. The computer application Siftrunner was used to run games on the set of cubes. In November 2012, Sifteo, Inc. released the second generation of Sifteo cubes, known as the Sifteo Cubes Interactive Game System. The new system came with a base unit that stored games available to the cubes and had speakers for game audio, thus eliminating the use of a computer when running games. The base unit can still sync with the computer via USB cable so that new games can be installed from the desktop software. The second generation also removed the need for a charging station by powering the base and the cubes with AAA batteries, as opposed to the Lithium Polymer batteries used in the original Sifteo cubes. Sifteo, Inc. also released a new SDK for use with the new system.[2]

Figure 1: All of the Sifteo cubes sensor capabilities [2]

Figure 1 demonstrates the different possible events a Sifteo cube can recognize. Although the new Sifteo cubes are functionally similar to the original ones, they cannot be used together. The original Sifteo cubes will not sync with the new system's base unit, and the new cubes cannot run games through Siftrunner. Also, games designed on the old SDK will not work with the new cubes. The original Sifteo cubes can support gameplay with up to 6 cubes, while the new system doubles that amount and can support 12 cubes. Sifteo SPICE was designed for the original Sifteo cubes. Each original Sifteo cube contains:

- 32-bit ARM CPU
- 128x128 color TFT LCD
- 3-axis accelerometer
- 8MB Flash
- Lithium Polymer rechargeable battery
- 2.4 GHz wireless radio
- Sifteo's near field object sensing technology[2]

In addition to the Sifteo platform, this program utilizes Ngspice to run the SPICE analysis and Gnuplot for plotting purposes.

Ngspice is a mixed-level/mixed-signal circuit simulator. Its code is based on three open source software packages: Spice3f5, Cider1b1 and Xspice. Ngspice is part of gEDA project, a full GPL'd suite of Electronic Design Automation tools. [4]

Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986. [5]

V. Requirements

a. System Requirements

In order to run the Sifteo desktop software, computers must meet the following requirements:

Windows
2.0 GHz Intel Pentium 4 or faster processor
Windows XP SP3 with 512 MB of RAM, or
Vista/Windows 7 with 1GB of RAM

Mac
1.5 GHz or faster Intel Core processor
Leopard 10.5 or Snow Leopard 10.6 with 1GB of RAM

General
1024x768 or larger display
Available USB 2.0 port
200 MB disk space (500 MB recommended)
Internet connection for software download and setup

Developing games for Sifteo also requires the use of an IDE, such as MonoDevelop, which is designed for C# and other .NET languages. Downloading the original Sifteo SDK provides the developer with a C# API, four demo programs, and documentation for the API and demos.[2]

b. Application Requirements

Sifteo SPICE was originally designed with freshman and sophomore level college students in mind, but it should be simple enough for anyone age 15 and up to configure and analyze a circuit without error. The application should utilize as many Sifteo sensor capabilities possible in

order to provide the user with as many intuitive options as possible. The graphics of the game should make use of the fullcolor LCD on each cube. Any text that appears in the game should be concise, easy to read, and easy to comprehend [2]. Sifteo SPICE was designed with a minimum cube requirement of 2 because any fewer and the circuit will no longer have anything to truly analyze. The maximum number of cubes Sifteo SPICE can utilize is 6, the maximum number possible with this build of the Sifteo Cubes.

VI. Design

Figure 2 shows a top-level system design which includes the user, the sifteo cubes, and the computer as the three most important components in this system. The user interacts with the cubes via the various events the cubes account for and the cubes relay this information to the computer via Bluetooth connection. This connection between the cubes and the computer is made possible through a provided USB link that allows Siftrunner to detect available cubes and their interactions. Figure 3 looks further into components within the computer and Sifteo cubes. The computer uses MonoDevelop and Siftrunner for the communication of the cubes. The computer also use the programs Ngspice and Gnuplot to run and output the SPICE analysis. The cubes contain a 32-bit ARM CPU, a 128x128 color TFT LCD, a 3-axis accelerometer, 8MB Flash memory, Lithium Polymer rechargeable battery, 2.4 GHz wireless radio, and near field object sensing technology. In order to play a game, you must run the program in MonoDevelop, load the game into Siftrunner, then select and run the app.

Figure 2: Top-level design of Sifteo SPICE

Figure 4 is a state diagram that depicts the process of building a circuit and running the analysis on that circuit. The initial cube is designated as the action cube responsible for running the analysis whenever the user wants. The action cube can be recognized among all the other cubes at the start of the application because it will be the only cube that is initially a voltage source. It is restricted to being the primary voltage source with preset input and output nodes (1 and 0, respectively). The input and output of the action cube will be represented by a green square as the input and a red square as the output. The initial input and output values of the other cubes will not be preset. The only thing that can change on this cube is the component value (voltage). Every other cube has the ability to change its component type and input/output node values until it is considered part of the circuit. When a cube is considered a part of the

circuit (it has a valid input and output node value), it loses the ability to have its component type changed but it gains the ability to have its component value adjusted. When the user is

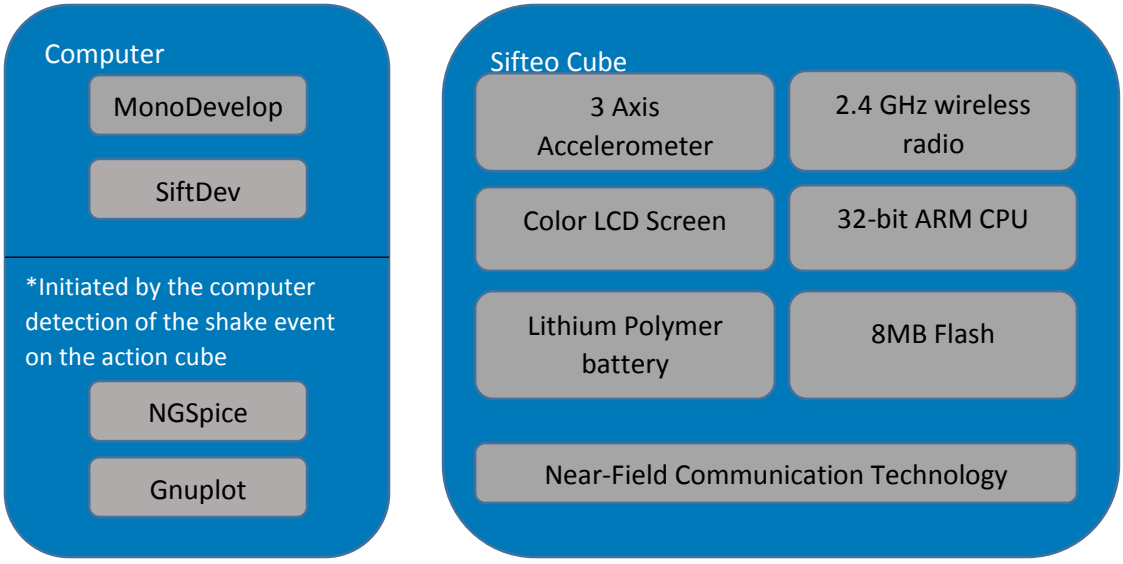

Figure 3: In-depth look at the system

satisfied with the created circuit, they simply have to shake the action cube to initiate the analysis. If the user is unhappy with their created circuit and want to start over, they can flip the action cube over to reset all the cubes (NOTE: This reset does not affect the input/output nodes or the component type of the action cube).

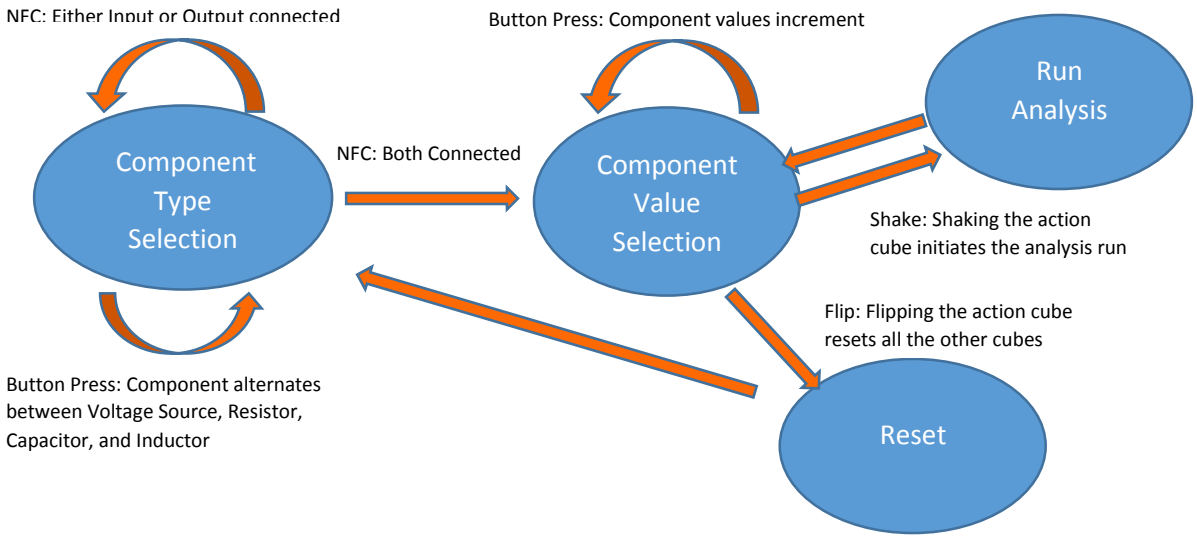

Figure 4: State flow diagram

VII. Development

The structured development process went the way I expected it to. If not for unforeseen circumstances I probably would have been able to implement even more functionality into this project. My development process started with the functionality and graphics of the sifteo cubes. After incorporating these aspects of the project I moved on to incorporating Ngspice and finally Gnuplot. Additional examples of what I found helpful throughout the development process can be found in Appendix D.

a. Functionality

I found that the best way to start developing Sifteo SPICE was to follow the tutorial written by Sean Voisen[6]. This tutorial provides the basic steps for creating simple Sifteo application, including installing an IDE and the SDK itself, creating and opening a project, using siftdev and siftulator to load your application and create graphics that satisfy Sifteo requirements, and running your application. Once I got this simple application up and running I looked through the sample games provided in the Sifteo SDK to find a game that was similar enough to what I was trying to accomplish. Luckiliy, the SlideShow app contained every event handler and a well-documented function for image drawing. The SlideShow app is a game that displays images and manipulates the graphics based on user interaction. I used this code as the basis for the Sifteo portion of my project. Also included in the SlideShow app was wrapper class for each individual cube. This was a very important discovery because this allowed me to store cube specific data which is important in displaying images and component values and creating netlists. The wrapper class also contained code that added every individual handler to each cube. The Sifteo cube object can designate handler functions for each button and accelerometer action. In Figure 5, we can see how we can add any necessary handlers to every cube. By assigning the OnButton function to cube.ButtonEvent, that function is called whenever the button press on the Sifteo cube is relayed back to the computer.

```
// Here we attach more event handlers for button and accelerometer actions.
      Cube mCube;
      mCube.ButtonEvent += OnButton;
      mCube.TiltEvent += OnTilt;
      mCube.ShakeStartedEvent += OnShakeStarted;
      mCube.ShakeStoppedEvent += OnShakeStopped;
      mCube.FlipEvent += OnFlip;
// ## Button ##
    // This is a handler for the Button event. It is triggered when a cube's
    // face button is either pressed or released. The `pressed` argument
    // is true when you press down and false when you release.
    private void OnButton (Cube cube, bool pressed)
    {
      if (pressed) {
        Log.Debug ("Button pressed");
      } else {
        Log.Debug ("Button released");
      }
    }
```

Figure 5: Example code for adding event handler functions

b. Graphics

The Image Helper tool is a feature of SiftDev, which converts user images into a Sifteo-friendly format. To convert a single file, the user clicks “Select File” and chooses the file from its directory. After selecting the file, the Image Helper displays it on the left side, under “Original”. Once you click “Convert”, the converted image will appear on the right side, under “Converted”, as shown in Figure 6. I used this process to implement all the component images and value text on the Sifteo cubes. Unfortunately, I encountered issues with my images not showing up on the cubes. The first pitfall invoved the SiftBundle not being created by the Image Helper. This problem was fixed by specifying the Bundle address by text instead of using the ‘Select Folder’ button. The other issue I encountered involved not specifying the images folder in the Siftulator program, seen in Figure 7. These minor errors caused a lot of delay because there wasn’t a reliable source of information regarding potential errors when dealing with the Sifteo platform.

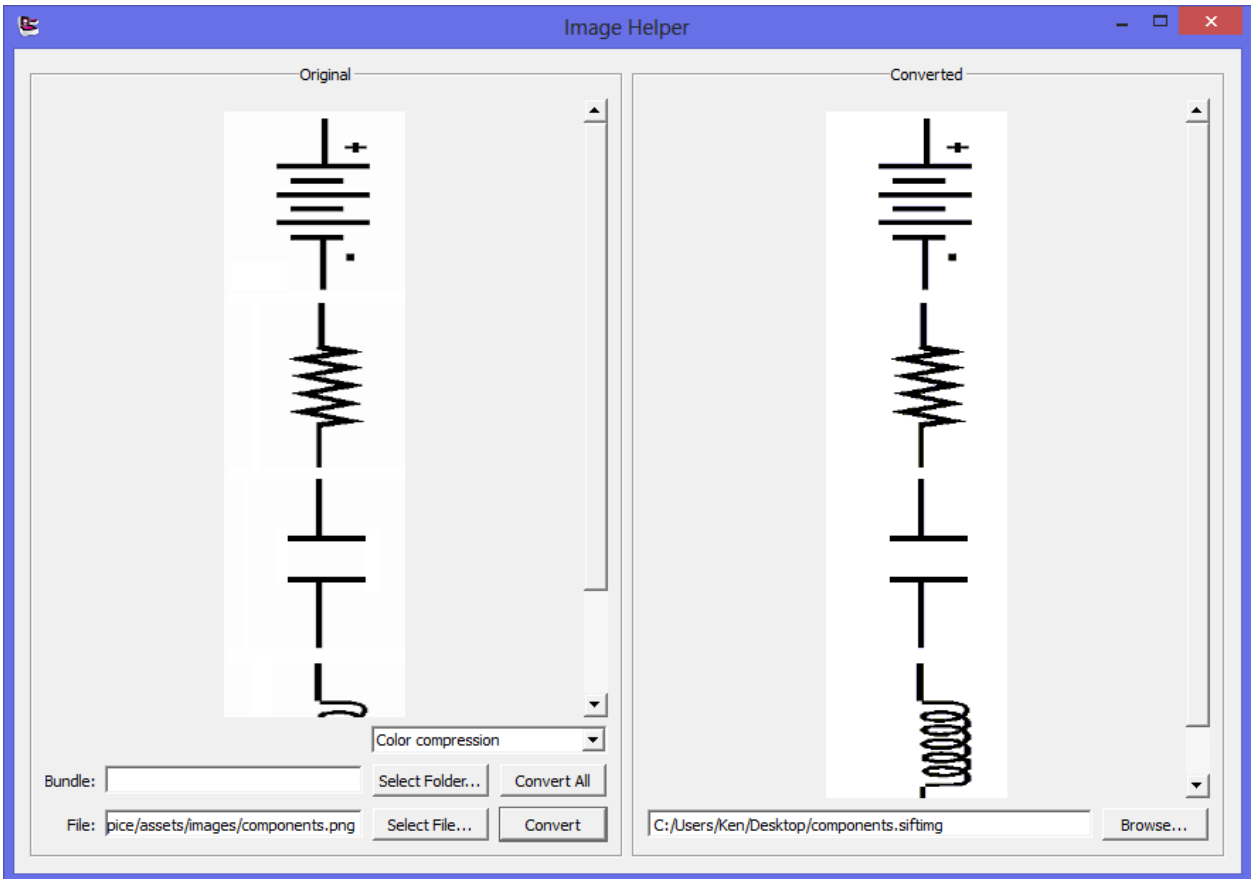

Figure 6: An example of Sifteo’s Image Helper tool

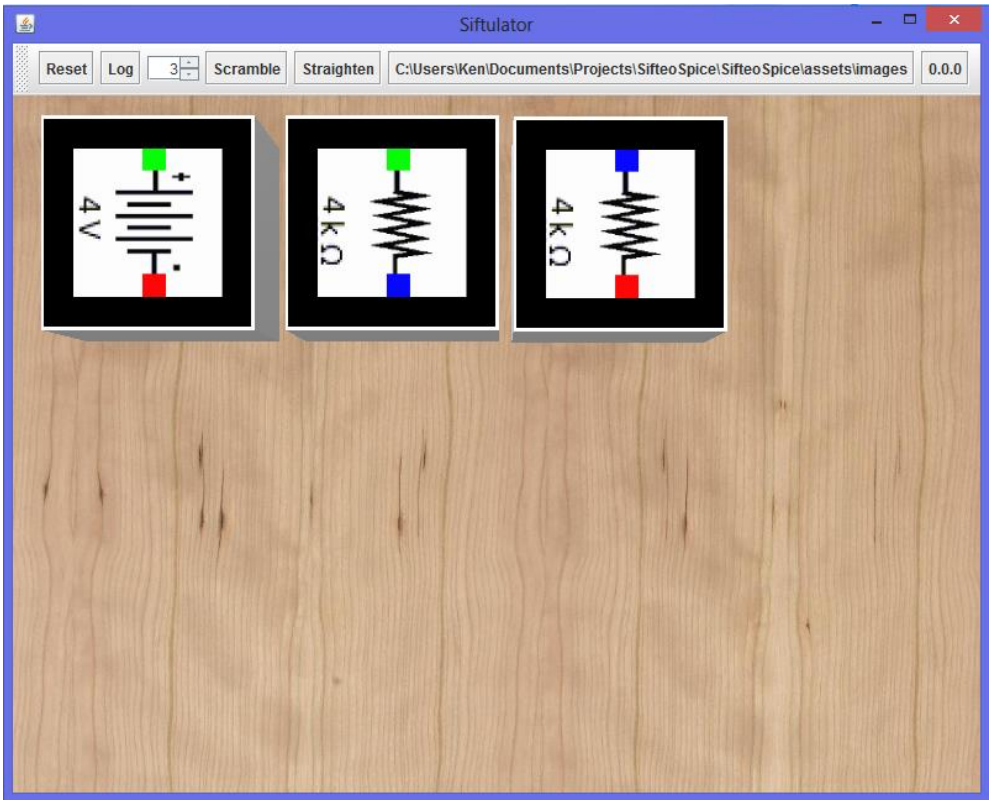

Figure 7: An example circuit on Sifteo Spice

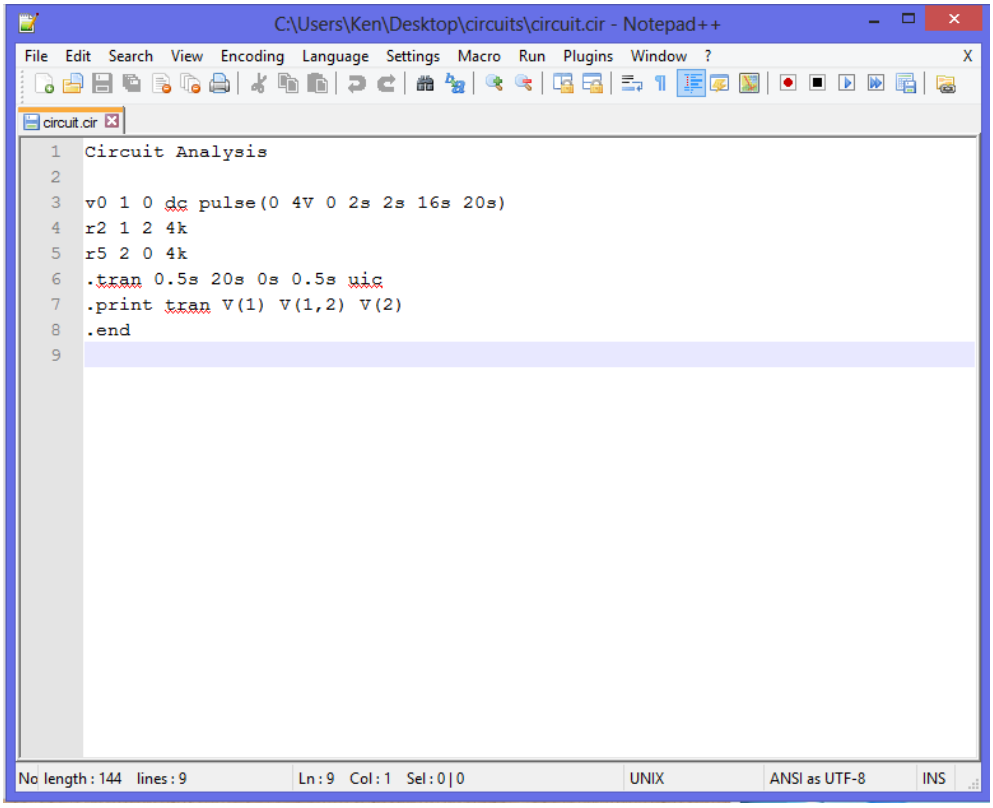

Figure 8: The corresponding netlist of the example circuit

c. Ngspice

Implementing Ngspice provided a minor challenge because it was the first time I tried to execute a separate process in C#. After some research I managed to find the appropriate function call to run this command. I had some trouble producing the correct results because I wasn't aware of certain case-sensitivity requirements by the program. Some parts of the netlist were not being recognized by the program which kept resulting in error. I initially printed everything in the netlist in uppercase because that how it was taught to me. I kept getting

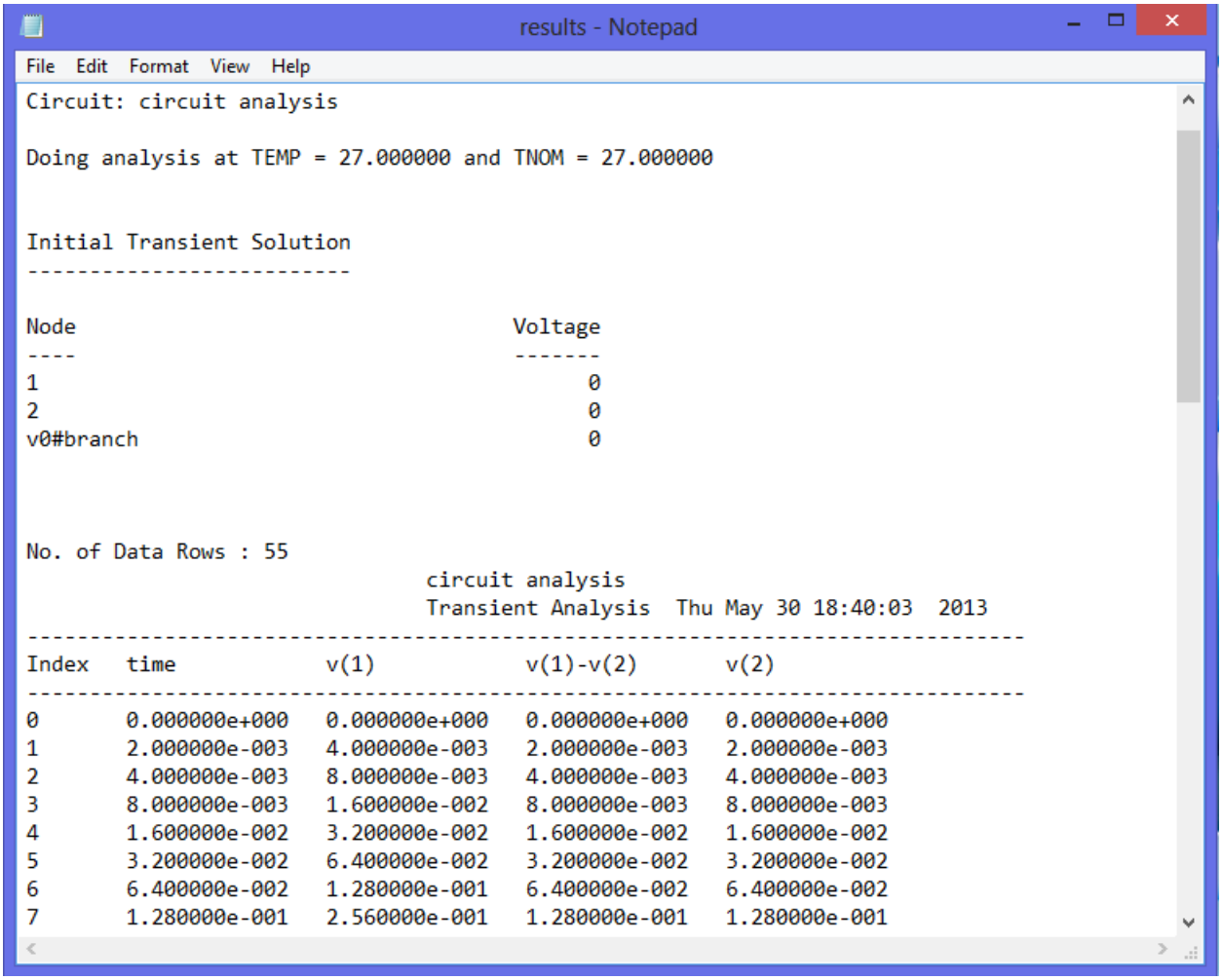

Figure 9: The corresponding SPICE analysis of the example circuit

weird errors and through some research I discovered that Ngspice had an issue with certain values being printed in uppercase. My solution, as seen in Figure 8, was to ensure that the netlist was entirely lower case, excluding the title which doesn't contribute to the analysis. The plotting program had an issue with a constant DC voltage value so I changed the analysis to a pulse to ensure that data would be available to produce a graph.

d. Gnuplot

Gnuplot was the hardest part to implement because it required additional commands to be input by a user and I had to reroute the standard input to my program, something I had never done in C# until then. This was solved by creating a StreamWriter object and setting it to standard input. After that it was easy to pass strings directly to standard input. After I solved this issue I encountered another problem involving the number of individual plots of data it could actually graph. For some reason this was limited so I was forced to limit the actual number of analyses I could do per circuit to the number of columns seen in Figure 9. Although index isn't used, the Voltage-Time graph seen in Figure 10 represents the coordinates seen in Figure 9.

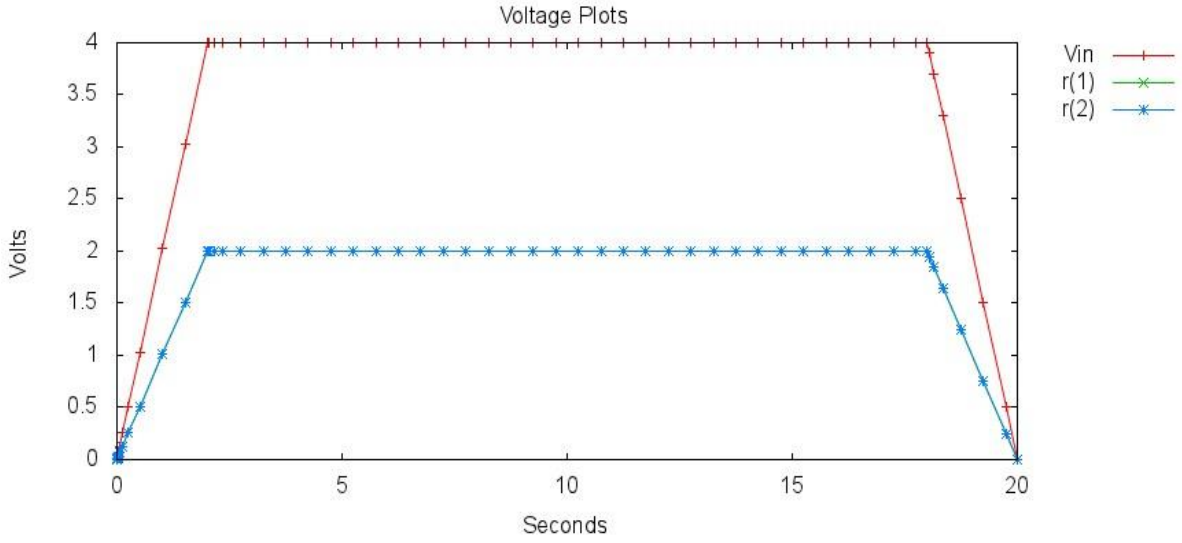

Figure 10: The Plotted output of the analysis

VIII. Testing

This project received ongoing testing throughout the development process and I made several significant observations along the way. The first thing to note is the availability of two testing platforms. Sifteo applications can be tested on either a physical set of Sifteo cubes or on Sifteo's own cube simulation program, Siftulator. Mickey's Memory Game was tested on both and I found it beneficial to do so. It was also important to ensure that the created netlist was correct and the corresponding plot was correct.

a. Siftulator vs. Sifteo Cubes

Siftulator is an application included in the Sifteo SDK, which allows developers to simulate cube functionality on the computer screen. It was an incredibly convenient tool, because it eliminated the need to bring the physical cube set around. Whenever changes are made to an application, Siftulator must install the updated game onto the physical cubes before enabling

play. For bigger applications with a lot of data, this can take several minutes. In testing, it was very useful to have changes visible in Siftulator within seconds.

When using Siftulator, it can be easy to forget some of the basic rules for running games. The biggest issues that came up were in regards to loading the images in the Siftulator. The Siftulator folder must always point to the /assets/images directory of the running program. If you are running a program that does not correspond to the Siftulator's image folder, you will receive no warning. Another rule when developing is that whenever a change is made to the assets folder of a project (i.e. adding a sound file, renaming an image), the game must be reloaded into Siftdev by going to the Developer menu and selecting "Load Apps". To load an app, you must select the folder that holds the assets folder, otherwise it will not show up in your "my games" section. [2]

b. SPICE Testing

The first part of the SPICE testing involved inspecting the resulting netlist file generated by the program. Some of the important things I had to consider regarding the netlist was the naming convention of the components and how I wanted to handle the voltage input. For the naming convention, I decided to include the first letter of the component type and cube ID. So for example, the action cube will always be a voltage source and have a cube ID of 0 so its name would be v0.

The voltage input issue was more difficult to solve because there are multiple factors that contribute to it. After initial testing I realized that giving the circuit a constant DC voltage yielded a good output file but Gnuplot wouldn't generate the plot. To solve this issue I decided that the initial voltage source would be a pulse voltage to ensure that a plot is generated. This ended up causing another issue regarding the analysis of capacitors and inductors. For those components to show useful data (as a learning tool) the voltage pulse should incorporate the RC or RL time constant. Unfortunately, I didn't have enough time to figure this out but it is definitely something that should be implemented in future work.

c. Test Cases

The following table depicts the test cases for the Sifteo and SPICE portions of this project:

Category	Test Cases
Sifteo	• All the relevant sensor events are handled correctly without breaking the images or the game • The maximum number of connections is never exceeded • Selecting the component types/values works as intended

	• Activating the analysis works, and only relevant cubes make it to the netlist • Resetting the cubes doesn't result in error
SPICE	• Netlist is generated without error • Ngspice is executed and the resulting output contains relevant data • Gnuplot is able to take the resulting data produced by Ngspice to create the correct plot

IX. Conclusion

This project provided a special challenge for me. Not only was I dealing with brand new language, I had to use that language to implement an application on a platform I had no prior experience with. I felt it was a very good lesson in versatility and being able to adapt quickly even though a lot of the information you might need is nearly impossible to find and the only hope is to learn by experience. The early part of this project was frustrating because of the time it took to finally get acclimated to platform and the language but it definitely paid off. The time I spent with the tutorials and sample games was very important and it was beneficial especially when certain circumstances required me to overhaul this project. I was fortunate enough to not encounter any real bugs regarding the platform because the amount of help available for any of them would be scarce if not non-existent.

The biggest reason for this was probably the fact that this new technology was constantly evolving during the course of this project. This made developing for Sifteo cubes tricky because search results usually resulted in newer iterations of the product that varied in their differences from the Sifteo cubes I used for this project. One big piece of advice I have for anyone hoping to develop for this iteration of the Sifteo cubes is to learn from the sample games and docs provided in the Sifteo SDK. These were very useful for me and I think the base code provided is enough for anyone to get to comfortable level with working on these cubes.

a. Future work

- *Create a graphical user interface on the computer in conjunction with the sifteo cubes*

  I think it would be more beneficial if the plots would transition when a user runs multiple analyses. It is also important to include a way to change the type of voltage source (AC, DC, Pulse) and I think it will be easier to do that in a GUI rather than on the cubes. We could also make Sifteo SPICE even more of a learning tool by having an option to display the current netlist so users can get more familiar with creating those.

- *Add a greater variety of analysis*
  Unfortunately, I only had enough time to implement analysis on the voltage across components. I think current analysis is also necessary for this project. I also think the RC and RL time constants should be included in the analysis if the voltage source is either pulse or AC voltage.
- *Add a greater variety of components and values*
  The goal of this project was to incorporate up to second-order circuitry but I think more complicated circuitry would be beneficial in making Sifteo SPICE an all-encompassing circuit learning tool. I also think more relevant component values should be added so students can create circuits they might encounter on a homework assignment or in a lab.
- *Graphics/Sounds Improvements*
  As of right now, this project doesn't incorporate sound and the graphics can and should be improved upon. The action cube also needs some sort of indicator, either by a graphic or a sound
- *Bug fixes and usability*
  Due to time constraints, there were some known bugs (and probably unknown ones) that weren't fully fixed that should be. Any attempt to make Sifteo SPICE more user-friendly is definitely welcome

Appendices

Appendix A: References

[1] Cordon, Karina. "Mac's Fiesta: A Foreign Language Game for the Sifteo Platform." June 2012. Web. < http://digitalcommons.calpoly.edu/cpesp/59/ >

[2] Concepcion, Anjelica. "Mickey's Memory Game: A children's memory game designed for Sifteo cubes." March 2013. Web. <http://digitalcommons.calpoly.edu/cpesp/80/>

[3] Peters, Kevin. "TOUCHSPICE: PHYSICAL-VIRTUAL CIRCUIT EMULATOR" June 2012. Web. <http://digitalcommons.calpoly.edu/theses/769/>

[4] Ngspice. Web. <http://ngspice.sourceforge.net/presentation.html>

[5] Williams, Thomas. Kelley, Colin. "gnuplot 4.6" 2012. Web. <http://www.gnuplot.info/docs_4.6/gnuplot.pdf>

[6] Voisen, Sean. "Up and running with the Sifteo SDK." October 2011. Web. < http://sean.voisen.org/blog/2011/10/up-and-running-with-sifteo-sdk/>

Appendix B: Senior Project Analysis

Summary of Functional Requirements

Sifteo SPICE is an application that can be run using with 3 or more Sifteo cube. It starts with each cube unattached to the circuit (except the action cube which is the only cube that has

default input/output node values). When separated from the circuit, any cube can change its component type by pressing the button on the screen. Each cube can also set their input/output nodes via near-field communication with other cubes. After a cube is connected to the circuit with input and output node values they can change their component values by pressing the button on the screen again. The analysis can be run when a user shakes the action cube and all the cubes can be reset by flipping the action cube. As of right now the action cube can only be recognized by the fact that it is the first voltage source seen when the program is started. An indicator of the action cube is something that is designated for future work.

Primary Constraints

The biggest constraint was the lack of readily available resources for help with the platform. This made developing on these cubes exciting because there wasn't really anything to refer to. Another big constraint for me was dealing with all the various options for Gnuplot. Getting a good graph wasn't easy and I couldn't figure out why there was a restriction on the number of data plots available for the program.

Economic

The cost of the 6 Sifteo cube set is about $250. For this project, no additional money was spent, as the cubes were previously purchased for another project. The Sifteo software is free to download and the

MonoDevelop IDE is free. If MonoDevelop is unavailable, it may cost an additional amount to download an alternate IDE, such as Visual Studio [2]. Luckily, I was able to develop on a different IDE called Xamarin Studio for the end of this project. The span of this project was 33 weeks, with approximately 2-12 hours of research or development done each week.

Environmental

The biggest environmental impact related to this project is the need to keep the batteries for all 6 cubes charged. Also, the game cannot be played with the cubes alone, it always requires the use of a computer to run the game.

Manufacturability

No manufacturing was done for this project.

Sustainability

The entire project can easily be passed on to others who wish to continue the development of the game. As long as someone has a set of Sifteo cubes and a computer that can run the Sifteo software, they can play the game. However, a newer Sifteo SDK was released in October 2012, which supports C++ projects instead of C#. Although the website still says that the old SDK is available for download, it is harder to find and it does not seem to be supported anymore.

Health and Safety

Extended continuous use of this application could cause strain on the eyes and possibly the hands. It is important to take breaks when engaging in use.

Development

This project was my first exposure to coding in C# and developing an MVC project. I also learned a lot from using the Sifteo API. Although this task may have seemed daunting when I first started, it helped me realize my potential to adapt to new programming situations.

Appendix C: Component Images

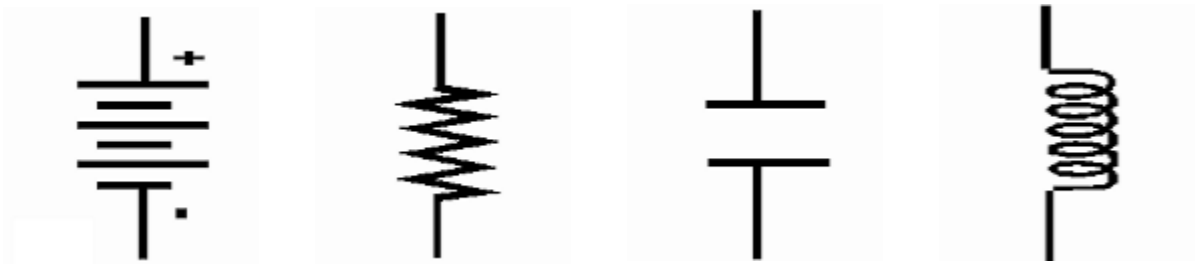

Figure 10: Images of the available circuit components

Appendix D: Tips for possible Issues

To get started I would suggest following Sean Voisen's Up and Running with the Sifteo SDK. http://sean.voisen.org/blog/2011/10/up-and-running-with-sifteo-sdk/ Chances are you may run in to some issues with this tutorial. Fortunately I have some solutions to pitfalls I encountered that should provide you with at least a base program that you can run on your Sifteo cubes.

- *I can't seem to find/download Monodevelop*
  You can download and install Xamarin Studios instead. It should work the same or at least very similarly because both IDE's are from the same company.
- *The mono function isn't generating anything*
  This function isn't absolutely necessary but I would first try to get it to work. If it doesn't work my suggestion is to just create a new solution using the Xamarin Studios 'New Solution' selection. (NOTE: you probably need to add certain files and folders to make sure this works with Sifteo. What I did was copy the files and folders from an existing sample game to ensure I wasn't missing anything).

Appendix E: Source Code

Bootstrap.cs

```
// When debugging, it can sometimes be useful to launch your app directly,
// outside of Siftrunner's app harness. Because BaseApps are meant to be run
```

```
// inside Siftrunner, a bootstrapping process is required to kick off the
// program manually.


// All the classes in your app should share a namespace.
namespace SifteoSpice
{

  // The Bootstrap class is a simple wrapper with a Main() method that kicks
  // off your app. All of your app's logic should go into the app class.
  public class Bootstrap
  {

    public static void Main(string[] args)
    {
      // Create the app and start it up.
      (new SifteoSpiceApp()).Run();
    }
  }
}
```

Main.cs

```
//Sifteo Spice
// Includes all the sifteo cube and spice functionality
// -------------------------------------------------------------------------
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using Sifteo;

namespace SifteoSpice
{
  public class SifteoSpiceApp : BaseApp
  {

    public String[] mImageNames;
    public List<CubeWrapper> mWrappers = new List<CubeWrapper> ();
    public Random mRandom = new Random ();
    public int connections;
    public String[] compTypes = {"v", "r", "c", "l"};

    // Here we initialize our app.
    public override void Setup ()
    {
      int i = 0;
      connections = 0;
      // Load up the list of images.
      mImageNames = LoadImageIndex ();

      // Loop through all the cubes and set them up.
      foreach (Cube cube in CubeSet) {
```

```
        // Create a wrapper object for each cube. The wrapper object allows
us
        // to bundle a cube with extra information and behavior.
        CubeWrapper wrapper = new CubeWrapper (this, cube);

        if (i == 0) { //The voltage source
          wrapper.mCubeIn = 1;
          wrapper.mCubeOut = 0;
          connections = 2; // The next available connection
          wrapper.mCubeState = 1; //State 1 is selecting the value
          wrapper.mCubeType = 0;
          // The first cube will always be a volt source (cannot change)
        }

        wrapper.mCubeId = i++;
        mWrappers.Add (wrapper);
        wrapper.DrawSlide ();
      }

      // ## Event Handlers ##
      // Objects in the Sifteo API (particularly BaseApp, CubeSet, and Cube)
      // fire events to notify an app of various happenings, including
actions
      // that the player performs on the cubes.
      //
      // To listen for an event, just add the handler method to the event.
The
      // handler method must have the correct signature to be added. Refer to
      // the API documentation or look at the examples below to get a sense
of
      // the correct signatures for various events.
      //
      // **NeighborAddEvent** and **NeighborRemoveEvent** are triggered when
      // the player puts two cubes together or separates two neighbored
cubes.
      // These events are fired by CubeSet instead of Cube because they
involve
      // interaction between two Cube objects. (There are Cube-level neighbor
      // events as well, which comes in handy in certain situations, but most
      // of the time you will find the CubeSet-level events to be more
useful.)
      CubeSet.NeighborAddEvent += OnNeighborAdd;
      CubeSet.NeighborRemoveEvent += OnNeighborRemove;
    }

    // ## Neighbor Add ##
    // This method is a handler for the NeighborAdd event. It is triggered
when
    // two cubes are placed side by side.
    //
    // Cube1 and cube2 are the two cubes that are involved in this
neighboring.
    // The two cube arguments can be in any order; if your logic depends on
    // cubes being in specific positions or roles, you need to add logic to
    // this handler to sort the two cubes out.
    //
    // Side1 and side2 are the sides that the cubes neighbored on.
```

```
    private void OnNeighborAdd (Cube cube1, Cube.Side side1, Cube cube2,
Cube.Side side2)
    {
      //This Function handles the input/output node selection between cubes
      Log.Debug ("Neighbor add: {0}.{1} <-> {2}.{3}", cube1.UniqueId, side1,
                 cube2.UniqueId, side2);

      CubeWrapper wrapper = (CubeWrapper)cube1.userData;
      CubeWrapper wrapper2 = (CubeWrapper)cube2.userData;
      //If either one is null then we have an illegal cube
      if (wrapper != null && wrapper2 != null) {
        // Cube.Side is an enumeration (TOP, LEFT, BOTTOM, RIGHT, NONE). The
        // values of the enumeration can be cast to integers by counting
        // counterclockwise:
        //
        // * TOP = 0
        // * LEFT = 1
        // * BOTTOM = 2
        // * RIGHT = 3
        // * NONE = 4

        if (side1 == Cube.Side.TOP) {
          if (side2 == Cube.Side.TOP) {
            if (wrapper.mCubeIn >= 0)
              wrapper2.mCubeIn = wrapper.mCubeIn;
            else if (wrapper2.mCubeIn >= 0)
              wrapper.mCubeIn = wrapper2.mCubeIn;
            else {
              //If both havent been assigned, make sure cube is part of the
              //circuit before we assign a new connection
              if (wrapper.mCubeOut >= 0 || wrapper2.mCubeOut >= 0) {
                wrapper.mCubeIn = connections++;
                wrapper2.mCubeIn = wrapper.mCubeIn;
              }
            }
          } else if (side2 == Cube.Side.BOTTOM) {
            if (wrapper.mCubeIn >= 0)
              wrapper2.mCubeOut = wrapper.mCubeIn;
            else if (wrapper2.mCubeOut >= 0)
              wrapper.mCubeIn = wrapper2.mCubeOut;
            else { //If both havent been assigned
              if (wrapper.mCubeOut >= 0 || wrapper2.mCubeIn >= 0) {
                wrapper.mCubeIn = connections++;
                wrapper2.mCubeOut = wrapper.mCubeIn;
              }
            }
          }
        } else if (side1 == Cube.Side.BOTTOM) {
          if (side2 == Cube.Side.TOP) {
            if (wrapper.mCubeOut >= 0)
              wrapper2.mCubeIn = wrapper.mCubeOut;
            else if (wrapper2.mCubeIn >= 0)
              wrapper.mCubeOut = wrapper2.mCubeIn;
            else {
              //If both havent been assigned, make sure cube is part of the
              //circuit before we assign a new connection
              if (wrapper.mCubeIn >= 0 || wrapper2.mCubeOut >= 0) {
```

```
                wrapper.mCubeOut = connections++;
                wrapper2.mCubeIn = wrapper.mCubeOut;
              }
            }
          } else if (side2 == Cube.Side.BOTTOM) {
            if (wrapper.mCubeOut >= 0)
              wrapper2.mCubeOut = wrapper.mCubeOut;
            else if (wrapper2.mCubeOut >= 0)
              wrapper.mCubeOut = wrapper2.mCubeOut;
            else { //If both havent been assigned
              if (wrapper.mCubeIn >= 0 || wrapper2.mCubeIn >= 0) {
                wrapper.mCubeIn = connections++;
                wrapper2.mCubeOut = wrapper.mCubeIn;
              }
            }
          }
        }
        /* If both input and output are set then you can no longer change the
         * component type and you can only change the values of the component
*/
        if (wrapper.mCubeIn >= 0 && wrapper.mCubeOut >= 0) {
          wrapper.mCubeState = 1;
        }
        if (wrapper2.mCubeIn >= 0 && wrapper2.mCubeOut >= 0) {
          wrapper2.mCubeState = 1;
        }
        //wrapper.mRotation = (int)side1;
        wrapper.mNeedDraw = true;
        wrapper2.mNeedDraw = true;
      }

      Log.Debug ("Neighbor add: {0}: {1}, {2} \n {3}: {4}, {5}",
wrapper.mCubeId,
                 wrapper.mCubeIn, wrapper.mCubeOut, wrapper2.mCubeId,
                 wrapper2.mCubeIn, wrapper2.mCubeOut);
    }

    // ## Neighbor Remove ##
    // This method is a handler for the NeighborRemove event. It is triggered
    // when two cubes that were neighbored are separated.
    //
    // The side arguments for this event are the sides that the cubes
    // _were_ neighbored on before they were separated. If you check the
    // current state of their neighbors on those sides, they should of course
    // be NONE.
    private void OnNeighborRemove (Cube cube1, Cube.Side side1, Cube cube2,
Cube.Side side2)
    {
      Log.Debug ("Neighbor remove: {0}.{1} <-> {2}.{3}", cube1.UniqueId,
side1,
                 cube2.UniqueId, side2);
      /*
      Good to include even though it isn't doing anything for this
application
       */
    }
```

```
    // Defer all per-frame logic to each cube's wrapper.
    public override void Tick ()
    {
      foreach (CubeWrapper wrapper in mWrappers) {
        wrapper.Tick ();
        if (wrapper.mCubeId == 0) {
          if (wrapper.mCubeFlag == 1) {
            Log.Debug ("Time to execute");
            wrapper.mCubeFlag = 0;
            executeSpice (mWrappers);
          } else if (wrapper.mCubeFlag == 2) {
            Log.Debug ("Clear cubes");
            wrapper.mCubeFlag = 0;
            clearCubes (mWrappers);
          }
        }
      }
    }

    // ImageSet is an enumeration of your app's images. It is populated based
    // on your app's siftbundle and index. You rarely have to interact with
it
    // directly, since you can refer to images by name.
    //
    // In this method, we scan the image set to build an array with the names
    // of all the images.
    private String[] LoadImageIndex ()
    {
      ImageSet imageSet = this.Images;
      ArrayList nameList = new ArrayList ();
      foreach (ImageInfo image in imageSet) {
        nameList.Add (image.name);
        Log.Debug ("image is  hey {0}", image.name);
      }
      String[] rv = new String[nameList.Count];
      for (int i=0; i<nameList.Count; i++) {
        rv [i] = (string)nameList [i];
      }
      return rv;
    }

    private void clearCubes (List<CubeWrapper> wrappers)
    {
      //reset all the cubes (except the action cube)
      //Set next available connection back to 2
      connections = 2;
      foreach (CubeWrapper wrapper in wrappers) {
        if (wrapper.mCubeId != 0) {
          wrapper.mCubeIn = -1;
          wrapper.mCubeOut = -1;
          wrapper.mCubeType = 1;
          wrapper.mCubeState = 0;
          wrapper.mCubeValNdx = 0;
        } else {
          wrapper.mCubeIn = 1;
          wrapper.mCubeOut = 0;
          wrapper.mCubeValNdx = 0;
```

```
        }
        wrapper.mNeedDraw = true;
      }
    }

    private void executeSpice (List<CubeWrapper> wrappers)
    {
      String analysis = ".print tran";

      string[] components = new string[6];
      int comp_cnt = 0, i = 0;
      // Write the string to a file.
      System.IO.StreamWriter file = new System.IO.StreamWriter
("C:\\Users\\Ken\\Desktop\\circuits\\circuit.cir");
      //If we have a header then we should include it before we write the
actual netlist

      /* 1. Creating the Netlist */
      file.WriteLine ("Circuit Analysis\n");
      foreach (CubeWrapper wrapper in wrappers) {
        if (wrapper.mCubeState == 1) { //If the state is 1 then both the
input and output of the component are connected
          //If voltage source then we need to include the pulse.
          //This might change depending on if we add more voltage options
          analysis += (wrapper.mCubeOut == 0) ? (" V(" + wrapper.mCubeIn +
")") : (" V(" + wrapper.mCubeIn + "," + wrapper.mCubeOut + ")");
          components [comp_cnt] = compTypes [wrapper.mCubeType];
          comp_cnt++;
          if (wrapper.mCubeId == 0) {
            file.WriteLine ("{0}{1} {2} {3} dc pulse(0 {4} 0 2s 2s 16s 20s)",
                            compTypes [wrapper.mCubeType], wrapper.mCubeId,
                            wrapper.mCubeIn, wrapper.mCubeOut,
wrapper.mCubeVal);
          } else
            file.WriteLine ("{0}{1} {2} {3} {4}", compTypes
[wrapper.mCubeType],
                            wrapper.mCubeId, wrapper.mCubeIn,
wrapper.mCubeOut,
                            wrapper.mCubeVal);
        }
      }
      file.WriteLine (".tran 0.5s 20s 0s 0.5s uic");
      file.WriteLine (analysis);
      file.WriteLine (".end");
      file.Close ();

      /* 2. Run the ngspice program on the given netlist */
      ProcessStartInfo start = new ProcessStartInfo ();
      // Enter in the command line arguments, everything you would enter
      // after the executable name itself
      start.Arguments = "-b -o
C:\\Users\\Ken\\Desktop\\circuits\\results.txt" +
        "C:\\Users\\Ken\\Desktop\\circuits\\circuit.cir";
      // Enter the executable to run, including the complete path
      start.FileName = "C:\\Program Files\\spice\\bin\\ngspice.exe";
      // Do you want to show a console window?
      start.WindowStyle = ProcessWindowStyle.Hidden;
```

```
      start.CreateNoWindow = true;

      // Run the external process & wait for it to finish
      using (Process proc = Process.Start(start)) {
        proc.WaitForExit ();

        // Retrieve the app's exit code
        int exitCode = proc.ExitCode;
      }

      /* 3. Run gnuplot */
      Process p = new Process ();
      start = new ProcessStartInfo ();
      //start.Arguments = "--persist";
      start.FileName = "C:\\Program Files (x86)\\gnuplot\\bin\\gnuplot.exe";
      start.RedirectStandardInput = true;
      start.UseShellExecute = false;

      p.StartInfo = start;
      p.Start ();

      using (StreamWriter sw = p.StandardInput) {
        if (sw.BaseStream.CanWrite) {
          //Write gnuplot commands here
          sw.WriteLine ("set term jpeg size 875,400\nset output " +
            "\"C:/Users/Ken/Desktop/circuits/plot.jpg\"\n");
          sw.WriteLine ("set size 1,1\nset origin 0,0\nset key outside\n");
          sw.WriteLine ("set xlabel 'Seconds'\nset ylabel 'Volts'\nset " +
            "x2label 'Voltage Plots'\n");

          //initiate
          sw.WriteLine ("plot \\\n\"C:/Users/Ken/Desktop/circuits/results" +
            ".txt\" using 2:3 with linespoints axis x1y1 title 'Vin', \\");
          while (i != comp_cnt) {
            sw.WriteLine ("\"C:/Users/Ken/Desktop/circuits/results.txt\" " +
              "using 2:{0} with linespoints axis x1y1 title '{1}({2})', \\",
                          i + 4, components [i + 1], i + 1);
            i++;
          }
          sw.WriteLine ();
        }
      }

      Process.Start (@"C:\\Users\\Ken\\Desktop\\circuits\\plot.jpg");
    }
  }

  // ------------------------------------------------------------------------

  // ## Wrapper ##
  // "Wrapper" is not a specific API, but a pattern that is used in many
Sifteo
  // apps. A wrapper is an object that bundles a Cube object with game-
specific
  // data and behaviors.
  public class CubeWrapper
  {
```

```
    public SifteoSpiceApp mApp;
    public Cube mCube;
    public int mIndex;
    public int mXOffset = 0;
    public int mYOffset = 0;
    public int mScale = 1;
    public int mRotation = 0;
    public int mCubeId;
    public int mCubeIn = -1;
    public int mCubeOut = -1;
    public int mCubeState;
    public int mCubeType = 1;
    //0 for volt source, 1 for resistor, 2 for capacitor, 3 for inductor
    public int mCubeValNdx;
    public int mCubeFlag = 0;
    public string mCubeVal;
    public compValue mVals = new compValue ();

    // This flag tells the wrapper to redraw the current image on the cube.
    //(See Tick, below).
    public bool mNeedDraw = false;

    public CubeWrapper (SifteoSpiceApp app, Cube cube)
    {
      mApp = app;
      mCube = cube;
      mCube.userData = this;
      mIndex = 0;
      mCubeState = 0;
      /* Each cube state will determine what the toggle is
       * (component or value) */
      mCubeValNdx = 0;

      // Here we attach more event handlers for button and accelerometer
actions.
      mCube.ButtonEvent += OnButton;
      mCube.TiltEvent += OnTilt;
      mCube.ShakeStartedEvent += OnShakeStarted;
      mCube.ShakeStoppedEvent += OnShakeStopped;
      mCube.FlipEvent += OnFlip;
    }

    // ## Button ##
    // This is a handler for the Button event. It is triggered when a cube's
    // face button is either pressed or released. The `pressed` argument
    // is true when you press down and false when you release.
    private void OnButton (Cube cube, bool pressed)
    {
      if (pressed) {
        Log.Debug ("Button pressed 1");
      } else {
        Log.Debug ("Button released");

        // Advance the image index so that the next image is drawn on this
        // cube.
        this.mIndex += 1;
```

```
        /*if (mIndex >= mApp.mImageNames.Length) {
          mIndex = 0;
        }*/
        mRotation = 0;
        mScale = 1;
        mNeedDraw = true;

        switch (mCubeState) {
        case 0:
          mCubeType = (mCubeType + 1) % 4; //4 components, vs res cap and ind
          Log.Debug ("type is {0}", mCubeType);

          break;
        case 1:
          mCubeValNdx = (mCubeValNdx + 1) % 10; //10 values per component
          mCubeVal = mVals.values [mCubeType] [mCubeValNdx];
          Log.Debug ("Value is {0}", mCubeVal);
          break;
        }

      }
    }

    // ## Tilt ##
    // This is a handler for the Tilt event. It is triggered when a cube is
    // tilted past a certain threshold. The x, y, and z arguments are
filtered
    // values for the cube's three-axis acceleromter. A tilt event is only
    // triggered when the filtered value changes, i.e., when the
accelerometer
    // crosses certain thresholds.
    private void OnTilt (Cube cube, int tiltX, int tiltY, int tiltZ)
    {
      Log.Debug ("Tilt: {0} {1} {2}", tiltX, tiltY, tiltZ);
      /* Tilt isn't used in our program but its good to keep it here*/
    }

    // ## Shake Started ##
    // This is a handler for the ShakeStarted event. It is triggered when the
    // player starts shaking a cube. When the player stops shaking, a
    // corresponding ShakeStopped event will be fired (see below).
    //
    // Note: while a cube is shaking, it will still fire tilt and flip events
    // as its internal accelerometer goes around and around. If your game
wants
    // to treat shaking separately from tilting or flipping, you need to add
    // logic to filter events appropriately.
    private void OnShakeStarted (Cube cube)
    {
      Log.Debug ("Shake start");
      //When we shake the 0th cube (the battery) we will create the netlist
      //and run it with ngspice and gnuplot
      string path = Directory.GetCurrentDirectory ();
      Log.Debug ("{0}", path);
      if (mCubeId == 0) {
        mCubeFlag = 1;
      }
```

```
    }

    // ## Shake Stopped ##
    // This is a handler for the ShakeStarted event. It is triggered when the
    // player stops shaking a cube. The `duration` argument tells you
    // how long (in milliseconds) the cube was shaken.
    private void OnShakeStopped (Cube cube, int duration)
    {
      Log.Debug ("Shake stop: {0}", duration);
      mRotation = 0;
      mNeedDraw = true;
    }

    // ## Flip ##
    // This is a handler for the Flip event. It is triggered when the player
    // turns a cube face down or face up. The `newOrientationIsUp` argument
    // tells you which way the cube is now facing.
    //
    // Note that when a Flip event is triggered, a Tilt event is also
    // triggered.
    private void OnFlip (Cube cube, bool newOrientationIsUp)
    {
      if (newOrientationIsUp) {
        Log.Debug ("Flip face up");
      } else {
        Log.Debug ("Flip face down");
        //Set the flag to reset all cubes on the next tick
        mCubeFlag = 2;
      }
    }

    // ## Cube.Image ##
    // This method draws the current image to the cube's display. The
    // Cube.Image method has a lot of arguments, but many of them are
optional
    // and have reasonable default values.
    public void DrawSlide ()
    {

      //Define colors for each new connection (max =6 if we have 6 cubes
      Color red = new Color (Color.RgbData (255, 0, 0));
      Color green = new Color (Color.RgbData (0, 255, 0));
      Color blue = new Color (Color.RgbData (0, 0, 255));
      Color purple = new Color (Color.RgbData (153, 0, 153));
      Color brown = new Color (Color.RgbData (102, 51, 0));
      Color teal = new Color (Color.RgbData (0, 255, 255));

      Color[] colors = {red, green, blue, purple, brown, teal};
      String imageName = this.mApp.mImageNames [0];
      String numbers = this.mApp.mImageNames [1];

      //Offsets for printing the numbers
      int[] number_offsets = {
        6,
        40,
        74,
        108,
```

```
        142,
        176,
        210,
        243,
        277,
        311,
        378,
        409,
        443,
        477
      };
      int number_off = 40;
      int num_cnt = 0;
      // When specifying the image name, leave off any file type extensions
      // (png, gif, etc). Refer to the index file that ImageHelper generates
      // during asset conversion.
      //
      // If you specify an image name that is not in the index, the Image
call
      // will be ignored.

      // You can specify the top/left point on the screen to start drawing
at.
      int screenX = mXOffset;
      int screenY = mYOffset;

      // You can draw a portion of an image by specifying coordinates to
start
      // reading from (top/left). In this case, we're just going to draw the
      // whole image every time.
      int imageX = 0;
      int imageY = 0;

      // You should always specify the width and height of the image to be
      // drawn. If you specify values that are less than the size of the
image,
      // only the portion you specify will be drawn. If you specify values
      // larger than the image, the behavior is undefined (so don't do that).
      //
      // In this example, we assume that the image is 128x128, big enough to
      // cover the full size of the display. If the image runs off the sides
of
      // the display (because of offsets due to tilting; see OnTilt, above),
it
      // will be clipped.
      int width = 128;
      int height = 128;

      // You can upscale an image by integer multiples. A scaled image still
      // starts drawing at the specified top/left point, but the area of the
      // display it covers (width/height) will be multipled by the scale.
      //
      // The default value is 1 (1:1 scale).
      int scale = mScale;

      // You can rotate an image by quarters. The rotation value is an
integer
```

```
      // representing counterclockwise rotation.
      //
      // * 0 = no rotation
      // * 1 = 90 degrees counterclockwise
      // * 2 = 180 degrees
      // * 3 = 90 degrees clockwise
      //
      // A rotated image still starts drawing at the specified top/left
point;
      // the pixels are just drawn in rotated order.
      //
      // The default value is 0 (no rotation).
      int rotation = mRotation;

      // Clear off whatever was previously on the display before drawing
      //the new image.
      //mCube.FillScreen(Color.White);
      mCube.Image (imageName, screenX, screenY, imageX,
                   height * mCubeType, width, height, scale, rotation);

      if (mCubeIn >= 0) {
        mCube.FillRect (colors [mCubeIn], 60, 0, 20, 20);
      }
      if (mCubeOut >= 0) {
        mCube.FillRect (colors [mCubeOut], 60, 108, 20, 20);
      }

      if (mCubeState == 1) {
        if (mCubeValNdx > -1 && mCubeValNdx != 9) {
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [mCubeValNdx + 1], 20, 14, 2, 3);
          num_cnt++;
        } else if (mCubeValNdx == 9) {
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [1], 20, 14, 2, 3);
          num_cnt++;
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [0], 20, 14, 2, 3);
          num_cnt++;
        }
        switch (mCubeType) {
        case 0:
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [11], 20, 14, 2, 3);
          break;
        case 1:
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [10], 20, 14, 2, 3);
          break;
        case 2:
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [12], 20, 14, 2, 3);
          break;
        case 3:
          mCube.Image (numbers, screenX, number_off + (num_cnt * 20), 4,
                       number_offsets [13], 20, 14, 2, 3);
          break;
```

```
        }
      }
      // Remember: always call Paint if you actually want to see
      //anything on the cube's display.
      mCube.Paint ();
    }

    // This method is called every frame by the Tick in SifteoSpiceApp
    //(see above.)
    public void Tick ()
    {

      // You can check whether a cube is being shaken at this moment by
looking
      // at the IsShaking flag.
      /*if (mCube.IsShaking) {
        mRotation = mApp.mRandom.Next(4);
        mNeedDraw = true;
      }*/

      // If anyone has raised the mNeedDraw flag, redraw the image on the
cube.
      if (mNeedDraw) {
        mNeedDraw = false;
        DrawSlide ();
      }
    }

    public class compValue
    {
      public List<string[]> values = new List<string[]> ();

      public compValue ()
      {
        string[] voltVal = {
          "1V",
          "2V",
          "3V",
          "4V",
          "5V",
          "6V",
          "7V",
          "8V",
          "9V",
          "10V"
        };
        string[] resVal = {
          "1k",
          "2k",
          "3k",
          "4k",
          "5k",
          "6k",
          "7k",
          "8k",
          "9k",
          "10k"
```

```
        };
        string[] capVal = {
          "1uF",
          "2uF",
          "3uF",
          "4uF",
          "5uF",
          "6uF",
          "7uF",
          "8uF",
          "9uF",
          "10uF"
        };
        string[] indVal = {
          "1H",
          "2H",
          "3H",
          "4H",
          "5H",
          "6H",
          "7H",
          "8H",
          "9H",
          "10H"
        };

        values.Add (voltVal);
        values.Add (resVal);
        values.Add (capVal);
        values.Add (indVal);
      }
    }
  }
}

// ---------------------------------------------------------------------------
//
// SifteoSpiceApp.cs
//
//
```