Architecture and Rationale

Introduction

CircuitSolver is a mobile app that allows one to analyze the voltage across, current through and resistance of components in their hand-drawn circuit. It aims to eliminate the inconvenience (and time) of using in-app circuit builders and printable components kits when trying to analyze circuits.

This document includes the architecture and design using the 4+1 architecture model

Architecture and Rationale

The application consists of 3 major components:

1. The Android User Interface
2. Circuit component identification algorithms using the computer vision library OpenCV
3. Custom Circuit analysis classes written in Java

Android Studio

CircuitSolver's end users targets engineers and students who have cell phones that can be used to take pictures of their hand-drawn circuits. Given that Android is the most popular smartphone operating system globally (with 87.6% of the total worldwide smartphone OS market share in Q2 2016) we opted to build an Android app to meet the demands of our user demographic.

Android Studio was a natural choice since it is the official IDE for Android. Android Studio offers an Android specific development environment with common setup tasks such as Gradle streamlined. Other IDE's are available and arguable offer more development options. Our application will have many limitations including a) not supporting circuit pictures taken at obscure angles or poor light conditions b) no enhanced camera features c) only 4 circuit components identifiable d) simply storage e) static images. Our application's most complex component will be the image processing and since OpenCV is supported in Android Studio we chose Android Studio.

Android SDK

Android SDK uses a derivation of MVC known as the Model View Presenter architecture.

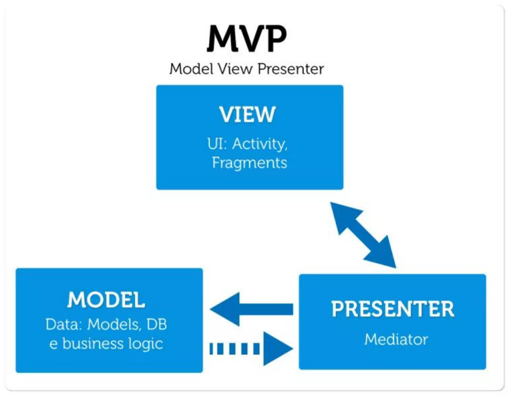


Image from: http://www.tinmegali.com/en/model-view-presenter-android-part-1/
The Android SDK implementation of MVP has the following features:

- UI is defined in an XML file
- Android Activity classes are extended to specify interaction between user and view
- Other classes can be created for handling business logic

OpenCV

OpenCV is a globally reputable computer vision library that is under a BSD licence. We also have experience using it and it has support for Android. Taken together there is no better library to build off of for the purposes of implementing algorithms that allows us to know the exact position of the corners and the components.

OpenCV offers a wide range of functions to work on an image and detect features. Since OpenCV added pre-implemented deep learning algorithms only recently and doesn't have the same reputation as other deep learning bases (see TensorFlow in the next paragraph), we are using OpenCv for all its well known and well-implemented non machine learning functions.

After using a classical Canny edge detection algorithm, a probabilistic Hough transform is performed on the input image. Using the hough transform with the least restrictive parameters (thresholds to 0, no line gaps,...) outputs a set of lines covering absolutely every edge detected by Canny, with all vertical lines being one nice line, and all other lines being an aggregation of chunks. After smoothing all the horizontal lines to make them one smooth line, we can make the following key observation: all lines being drawn that are neither horizontal nor vertically are just a cluster of small diagonal chunks. Seeing that a component is never all horizontal nor vertically drawn (The resistor has diagonal lines, the voltage source is a circle, the inductor has some arcs, and a capacitor has to be drawn slightly diagonal), we know that these clusters are the components.

Starting from this observation, the rest of the program is straightforward. We run a DBSCAN clustering algorithm on the the lines to find the position of the components, and keep the residual lines that are not part of a component for the corner detection. To detect the corners, we use the residual lines to find intersections or pseudo intersections (intersections within a certain threshold).

Finally, we know that a wire of a circuit is either horizontal or vertical. Based on this observation, we can use a recursive function to iterate over all corners and searching in all vertical and horizontal directions to create wires. Using this algorithm and some processing to have a well-defined circuit, the OpenCV service outputs all detected wires and component positions, but provides no information on which component is at which position.

TensorFlow

Due to the limitations within OpenCV, as well as the almost infinite possibilities of users drawing different components, we can't rely on pure image processing to distinguish components from one another. Therefore, we are using TensorFlow for distinguishing between components. When OpenCV detects a components it will take a section of the image containing the component and pass it into a pre-trained TensorFlow model for analysis.

Since TensorFlow is a very new library, being released just over a year ago, there are very few problems solved by the community. Included with the TensorFlow repository is an example Android application, using an example pre-trained model (a map of a neural network that identifies images). To generate the pre-trained model, we collectively drew hundreds of individual components to train the model.

After the model was shown to be accurate, we had to merge the example application with our application. Since TensorFlow is primarily written in C++, it had to be compiled with Bazel, a build tool by Google, to generate the required libraries. Then we moved the libraries into our application with the pre-trained model, and tied it in with OpenCV to detect specific components within the circuit.

Data Persistence

Circuit data is saved in the file system of the Android device. The only data persisted includes the:

- Original circuit image
- Circuit Schematic image **
- Downsized version of original circuit image
- OpenCV-processed image
- Circuit definition text file.

Strategy For Saving Circuits

- When the user creates a new circuit, whether through taking a photo or drawing, our app saves the circuit information to a text file, enabling the user to save and load the circuit again at a later time.

- The app does not write to the Circuit Definition file everytime the user makes an edit to their circuit. It only writes to the file when the user goes from one Activity to the Edit Circuit Activity, or from the Edit Circuit Activity to another Activity. To clarify, the app will only write to the Circuit Definition file when:
    - The user goes from the Take Photo screen to the Edit Circuit screen
    - The user goes from the Edit Circuit screen to the
- Each line in the Circuit Definition file represents an element in the circuit, with its corresponding coordinates and value

Circuit Definition File

Our first iteration of the Circuit definition file is based off of Hausen's open source circuit-simulator (https://github.com/hausen/circuit-simulator) and loosely resembles the well-known SPICE circuit simulator description language. Hausen's version is simpler for visually representing circuits, as it includes x,y coordinates for the position of elements in the circuit.

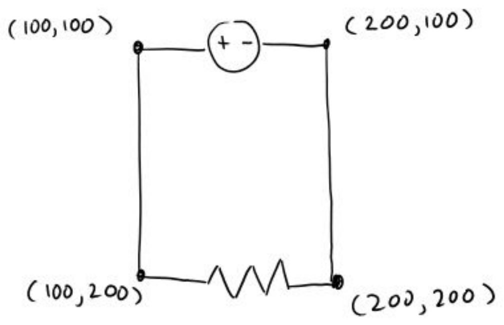
Specifications

Rows beginning with '$' indicate that the row contains meta data. So far the only meta data that our parser interprets is the original width and original height of the circuit display dimensions (which should be contained in the first row). This is useful for scaling the circuit size to fit the screen better. Each row after in the file represents a new element. The value in the columns follows this format: type, x1, y1, x2, y2, value. Wire elements should not have a value associated with them.

<file_name>.txt

```
$ 1000 1000
v 100 100 200 100 10.0
r 100 200 200 200 10.0
w 100 100 100 200
w 200 200 200 100
```

This file would represent the following circuit:

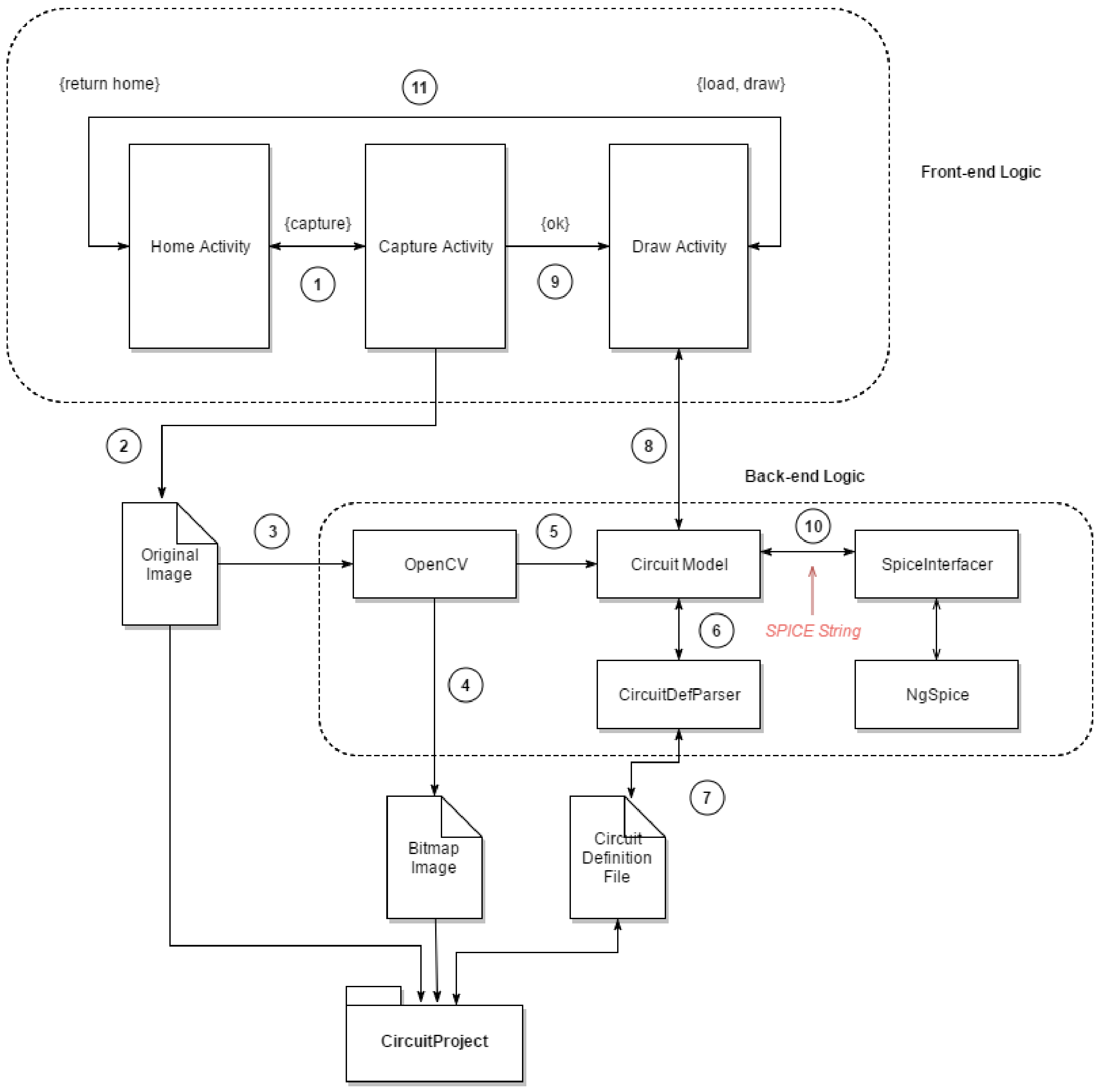


**The (x,y) pairs represent the coordinates

NgSpice

Our application relies on Ngspice to solve circuits. Nsgspice is an open source circuit simulator, written in C. By making small modifications to the source code, and using the Android NDK toolset, we were able to cross compile Ngspice for arm. Our app ships with a copy of Ngspice precompiled for ARM. We chose to use an already existing circuit simulator because of our time constraints. Ngspice allows us to easily add additional circuit features without worrying about circuit simulation techniques.

Dynamic View and Description



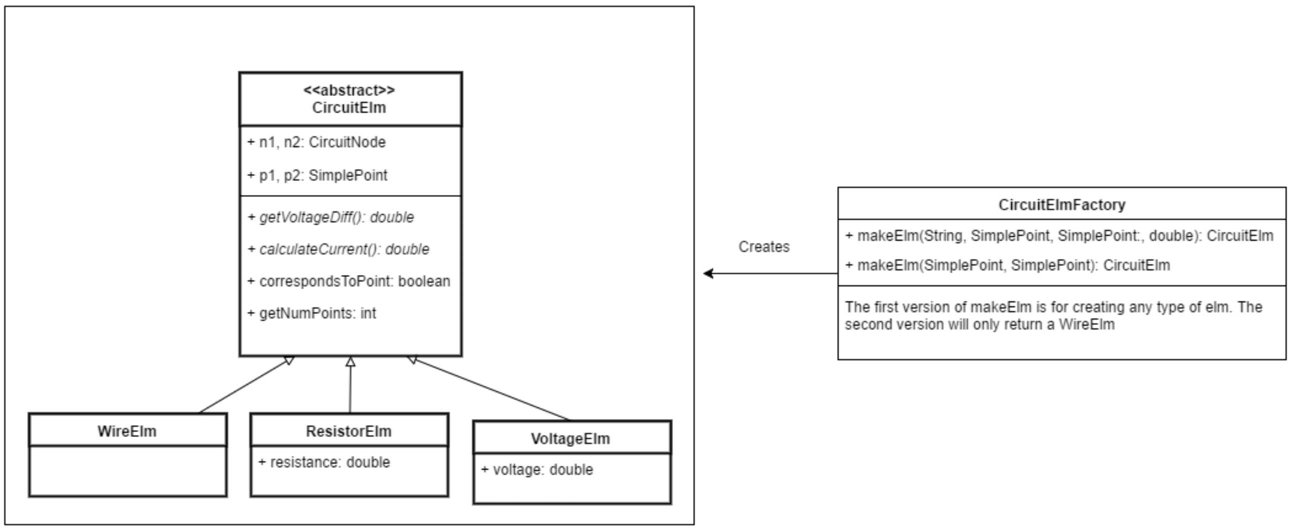
This diagram’s purpose is to provide a high-level understanding of the architecture, so some details are omitted or abstracted. The “front-end logic” components refer to the Android Activities our application. Our app consists of 3 Activities: Home Activity, Capture Activity, and Draw Activity. The back-end logic can be described as 4 main components: OpenCV, NgSPICE, Circuit Definition Parser, and Circsuit Model. The Circuit Model is composed of our custom CircuitElm classes and AnalyzeCircuit class.

(1) Going from Home view to capturing a photo via an external application. “Capture” here does not solely refer to using a camera application as a gallery application can also be used to retrieve an image from file. Developers could extend the capture activity to include other services without changing the Dynamic View (e.g. capturing an image from dropbox)
(2) After the user confirms a captured image, the original image will be saved on the file system
(3) The original image is also sent to OpenCV for processing
(4) OpenCV creates a processed bitmap displaying its interpretation of the circuit
(5) The processed bitmap from OpenCV is converted to a list of CircuitElm - a custom Java class we designed for representing circuit elements. The Draw Activity uses these classes to manipulate and solve the circuit.
(6) The list of circuit element objects is passed to the CircuitDefParser.
(7) CircuitDefParser parses the elements into a circuit definition text file, which is saved to the file system.
(8) Any manipulation of circuit or solving of circuit involves an interaction between these two components
(9) Application moves from Capture view to Draw view
(10) Request for analysis sent to NgSPICE component, which returns a result

Detailed Design

The following diagrams show a break down of the key components of the application.

Circuit Element Class Diagram


<<abstract>>
CircuitElm
+ n1, n2: CircuitNode
+ p1, p2: SimplePoint
+ getVoltageDiff(): double
+ calculateCurrent(): double
+ correspondsToPoint: boolean
+ getNumPoints: int
WireElm
ResistorElm
+ resistance: double
VoltageElm
+ voltage: double
Creates
CircuitElmFactory
+ makeElm(String, SimplePoint, SimplePoint:, double): CircuitElm
+ makeElm(SimplePoint, SimplePoint): CircuitElm
The first version of makeElm is for creating any type of elm. The second version will only return a WireElm

Note: Standard getters and setters exist for all the attributes in the circuit elements but are not displayed in UML to avoid crowdedness.

Circuit Analysis Dependency Diagram

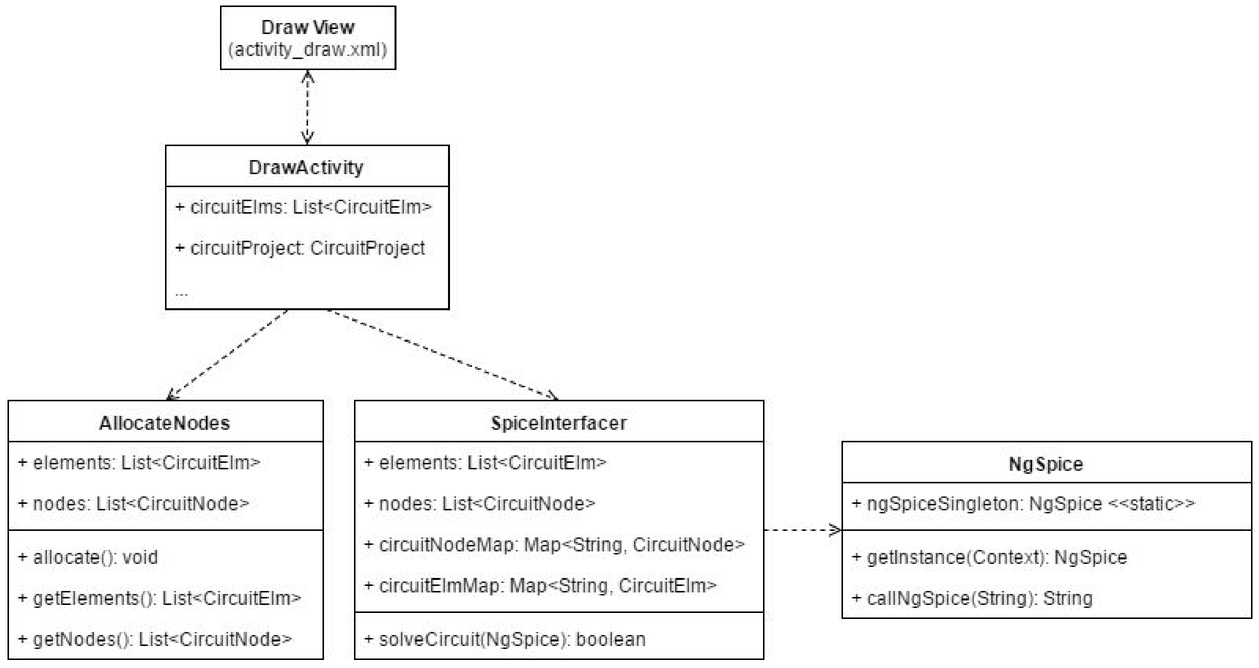
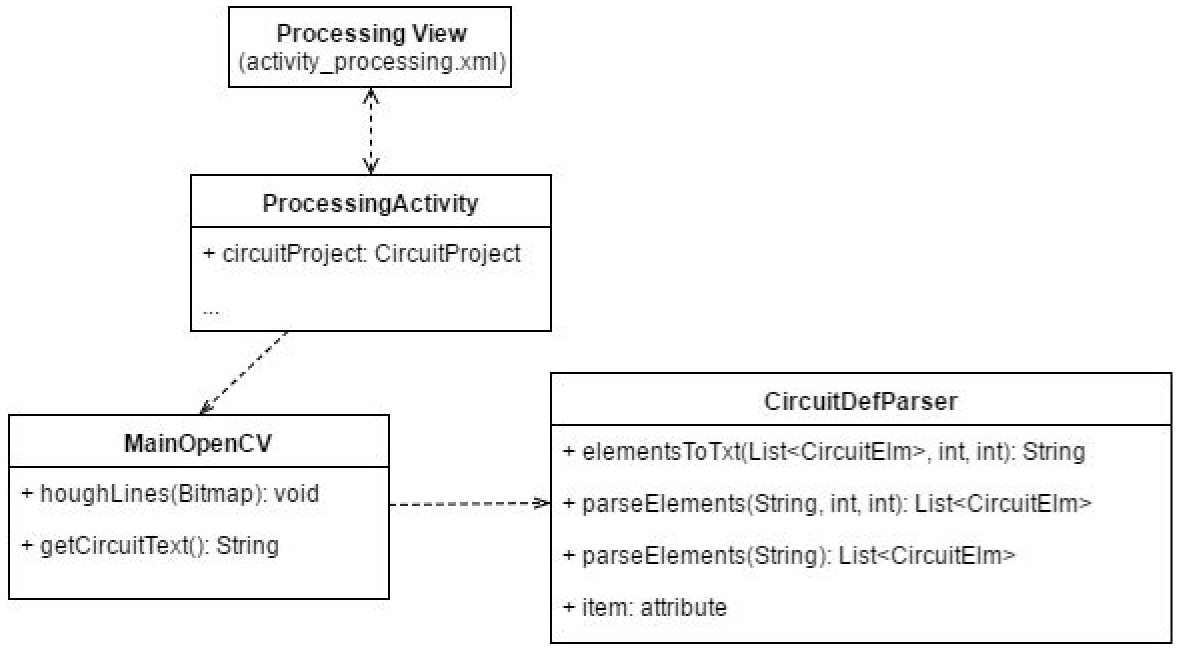

Draw View
(activity_draw.xml)
DrawActivity
+ circuitElms: List<CircuitElm>
+ circuitProject: CircuitProject
...
AllocateNodes
+ elements: List<CircuitElm>
+ nodes: List<CircuitNode>
+ allocate(): void
+ getElements(): List<CircuitElm>
+ getNodes(): List<CircuitNode>
SpiceInterfacer
+ elements: List<CircuitElm>
+ nodes: List<CircuitNode>
+ circuitNodeMap: Map<String, CircuitNode>
+ circuitElmMap: Map<String, CircuitElm>
+ solveCircuit(NgSpice): boolean
NgSpice
+ ngSpiceSingleton: NgSpice <<static>>
+ getInstance(Context): NgSpice
+ callNgSpice(String): String

Image Processing Dependency Diagram

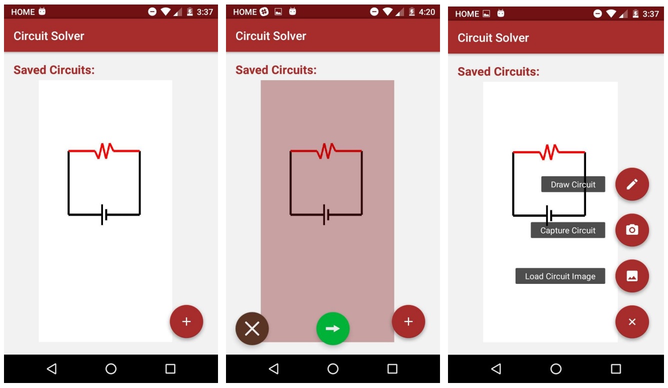

Processing View
(activity_processing.xml)
ProcessingActivity
+ circuitProject: CircuitProject
...
MainOpenCV
+ houghLines(Bitmap): void
+ getCircuitText(): String
CircuitDefParser
+ elementsToTxt(List<CircuitElm>, int, int): String
+ parseElements(String, int, int): List<CircuitElm>
+ parseElements(String): List<CircuitElm>
+ item: attribute

GUI

CircuitSolver’s user interface and experience is designed to be intuitive and simple to ease the learning curve for new users. We are using Android’s standard design package to design CircuitSolver. The app consists of one base activity and five separate activities or screens that all inherit from the base activity.

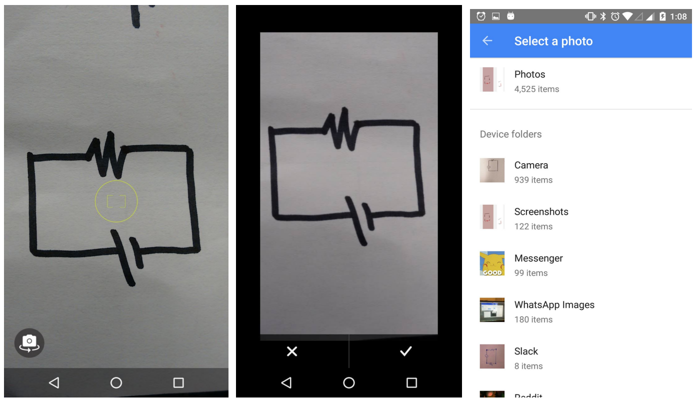
Home Screen

The home screen is where you would be able to see all the circuits that you have analyzed in the past and review and edit them further. There will also be a floating button in the bottom right of the screen to take you to either draw the circuit, take a picture, or load a circuit Image. Below are screenshots of what each screen will look like.



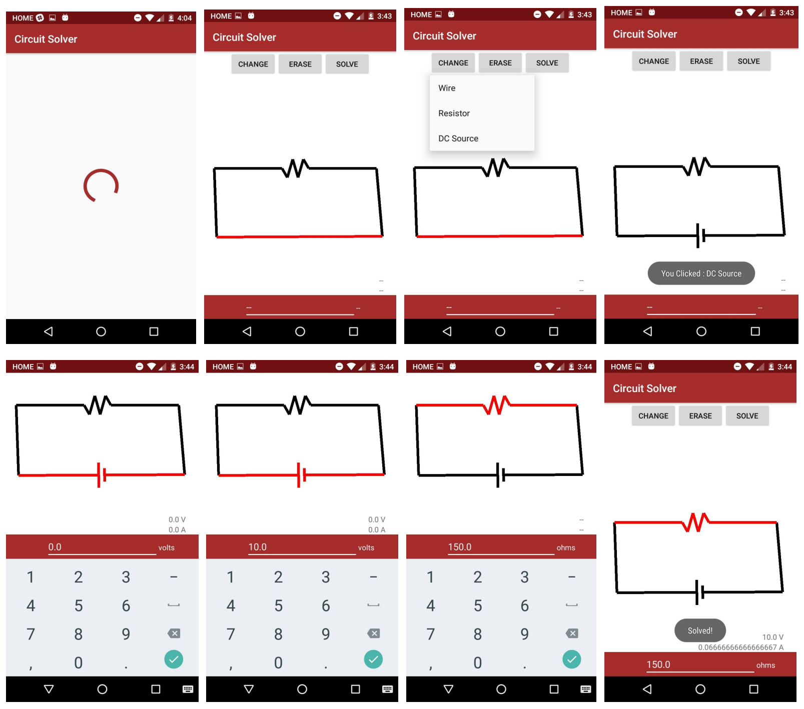
Camera Screen

The camera screen is where you have the ability to take a picture of your circuit schematic. It gets launched from the home screen, and the user interface is dependent on the version of the phone’s operating system as the app launches the default camera activity resulting in a varying look, but all will have the same ability. Once the picture is taken, the user will confirm the image looks good, and then be transferred to an edit circuit screen You may also choose to load a circuit image if that option was selected from the home screen.



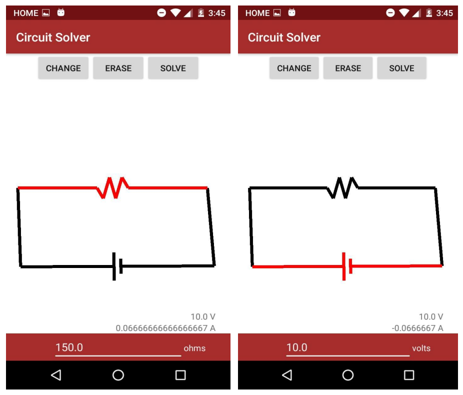
Edit Circuit Screen

The edit circuit screen begins with a loading screen, which will continue to spin until the circuit is done being processed. Once the circuit is finished being processed, the user will be able to correct or add to the system if a component was identified incorrectly. Once the user is satisfied with the values and components detected, the user will proceed to the analysis screen by pressing the "solve" button.



Analysis Screen

The analysis screen is the heart of the whole design. Every component mentioned was a stop in the process to get to this point. The user can now see the fully digitized circuit and can tap on any component to figure out the amount of current flowing through the component, as well as the voltage drop across the component.



Validation

Validation asks "Are we building the right product?"

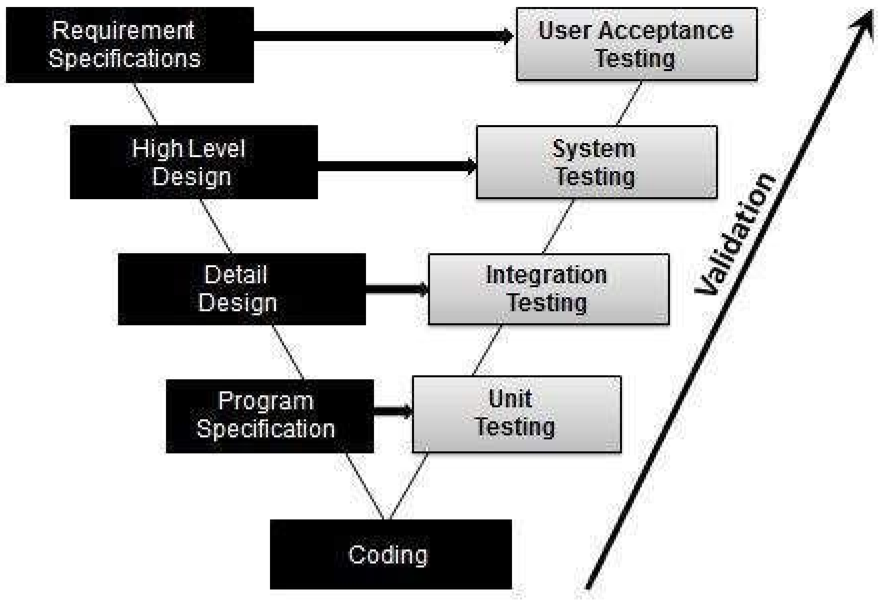


Image [source]

Validation will involve unit, integration, system and user acceptance (usability) testing. At the time of writing we have only implemented unit tests for circuit logic and received preliminary user acceptance test results. Validation testing represents the majority of work that will be implemented December 2 -December 13 (our final demo date).

Unit Testing
Unit tests that have already been are specified to test ngspice logic:

- NgSpiceTest: Calls Ngspice from within the Android app with a preset input net list. Compares Ngspice's output to the expected output.
- SpiceInterfacerAndroidTest: Tests the entire flow from a circuit described using our own representation, to a solved version of the circuit.

See the Developer Specific Information for more information.

Future unit tests will follow the correctness of data for all classes and methods (not getters nor setters) with the aim of 70%+ total coverage.

Integration and System Testing
The objectives of system testing is to validate the public exposed interfaces of the platform, by behaving as an end user, with no privileged access to internal systems. Integration verification

attempts to verify that separate systems operate well together before they're used in the whole system. Despite their clear different objective, in our approach Integration Testing and System Testing will not be explicitly separated.

This is because, for both, we will make use of Espresso, an extensible Android UI Test Framework provided by Google. As an instrumentation testing tool Espresso provides a set of hooks into the Android system that allow developers to control the lifecycle of Android components (i.e. drive the activity lifecycle instead of having these driven by the system). From a System Testing perspective it is an ideal approach since Espresso can simulate all use cases (see requirements documentation) step by step without accessing internal systems. Instead, the framework is designed to simulate a user's experience. Hamcrest matchers (which replace the deprecated MoreAsserts JUnit Assertion extension) and Espresso View Matchers (assertions about views which replace the deprecacated ViewAsserts) can be used to validate proper view layout and assert that particular UI elements are displayed, or have been interacted with etc. We have thoroughly specified five use cases (with branching alternate flows) in our requirements documentation

- USE CASE 1: Edit a previous saved circuit in order to change some components or change numerical values.
- USE CASE 2: View component values of a previous analyzed circuit
- USE CASE 3: Solve a circuit from an uploaded or taken circuit picture
- USE CASE 4: Create virtual circuit by directly drawing in the app
- USE CASE 5: Delete saved circuit

System testing will involve simulating each use case as an Espresso Test Class (with multiple sub-tests for alternate flows) and ensuring that, from the user's perspective, the expected output is achieved. This design for system testing will appropriately fall within the scope of black-box testing since it will allows for assertions without knowledge of the inner design of the code or logic.

For integration testing this approach is also ideal. Hamcrest matchers and Espresso View Matchers can be used explicitly to test integration among subsystems for use cases that see the user send information from one to the next since Hamcrest Matchers can also be used to validate sets, lists etc are behaving as intended given a backend/frontend process has occurred (e.g. test how how DrawActivity sends output to ngSpice, given particular processes occurred during our openCV service). Moreover, Espresso can do hermetic inter app testing (e.g. simulate a user fetching a bitmap using a camera) through basic usage of intents across application using mock objects, allowing for intent verification and stubbing. This further allows for integration testing for CircuitSolver's camera and load gallery functions. As such our approach sees an interface between system testing and integration testing. Instead of having separate classes for integration tests our simulated use cases will also see integration testing with assertions checking backend logic across individual software modules. This is white-box testing: access to subsystem internals is allowed.

Finally, it should be noted that data will have to be simulated. I.e. the same test may be performed on 10 different circuit examples. The code will be easily extensible so as to support the addition of example circuit images/circuit data to be used as test cases.

Usability Testing

In the pre-coding phases where requirements-program specification took up more of our focus we conducted usability testing through a continuous process in which a UI expert is consulted on a weekly/bi-weekly basis for evaluation and review. The consultant was shown our current version of the application and allowed to interact with the UI to provide feedback. Suggestions were recorded and implemented into the requirements or design as is deemed feasible.

Moving forward, through to the validation phase we wish to move more towards testing CircuitSolver's usability quantitatively on our actual target demographic. Our app is mainly for college students we want to solve circuits. Efficiency is the most important criteria for usability for CircuitSolver because it forms the bedrock assumption holding up that our product should even exist. If our product cannot offer users a more efficient/convenient means of solving circuits than other software then we have not built a product worth building. As such, our of the 3 usability criteria we would focus on measuring efficiency. This would be done simply through "hallway testing" where we would have clients - who have had no prior exposure to the app - perform tasks from our use cases. The time taken will be recorded and compared against the time taken to complete the same task via alternative/traditional methods (other apps and/or pen and paper and calculator). This way we will be able to ascertain a preliminary quantitative measure of effectiveness.

Developer Specific Information

Getting Started

These instructions assume you have Android Studio and Git installed as well as a github account. We recommend using your own device rather than the emulator.

How to find our source code:

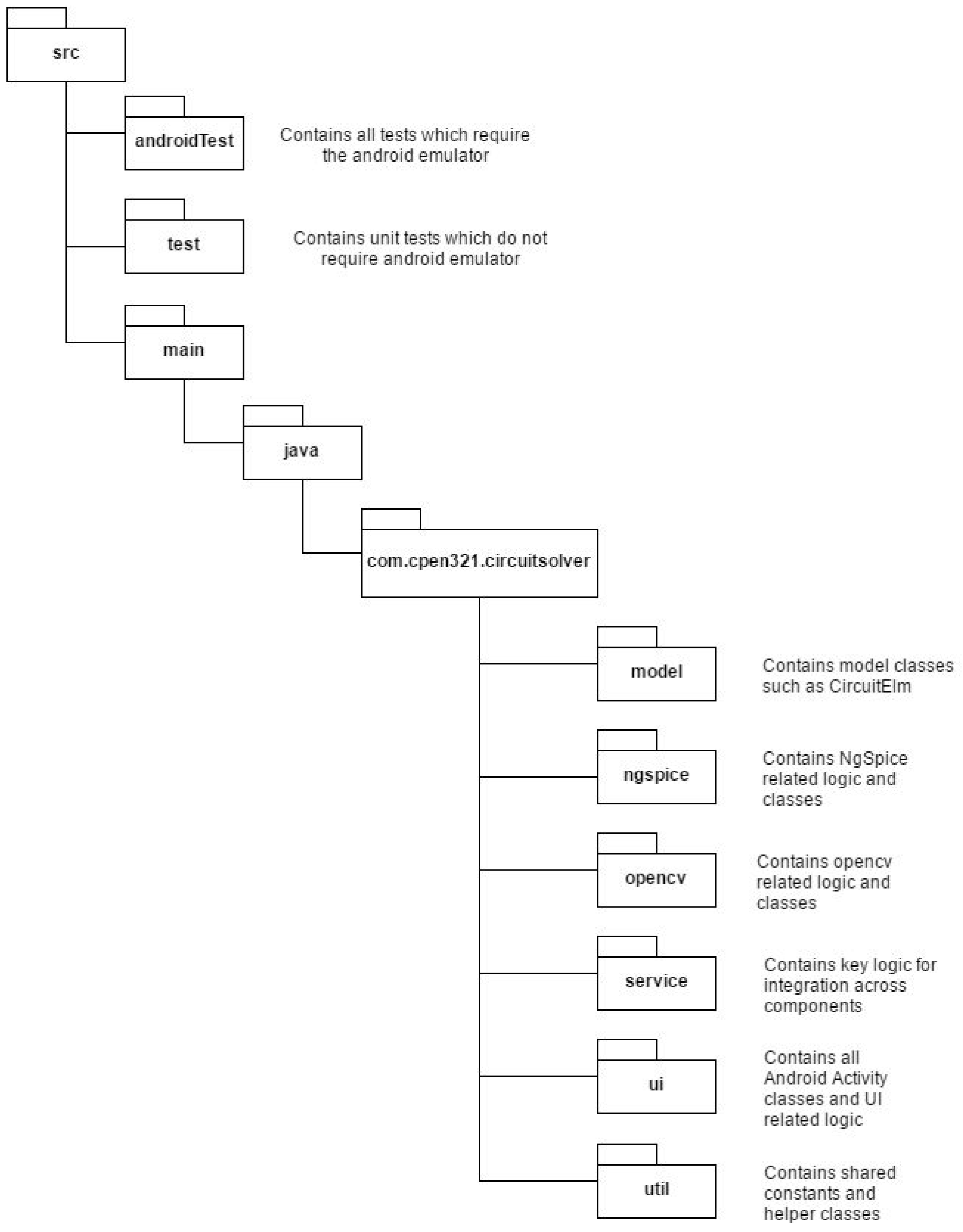
The source code can be found at, https://github.com/Frikster/CircuitSolverApp, In the src folder or https://github.com/Frikster/CircuitSolverApp/tree/master/app/src

How to check out and build the project:

1. Fork our repo on github to your own github account
2. In Android Studio go to VCS>Checkout from Version Control>GitHub (or when starting a new project select Checkout from Version Control>Github). Log into your github account.

3. Select the git repository URL that corresponds with CircuitSolverApp your desired parent directory and directory name. Be sure the parent directory exists - create folders/subfolders manually if needed.
4. You should see notifications the first time to “install missing platform(s) and sync project.” Press this to install the Android SDK platform. Next, “Install Build Tools and sync project” should appear for various versions. Install each one. If these options are already installed a notification simply asking you to sync with gradle should appear. Select this instead.
5. Go to tools > Android > Android SDK Manager > click “launch standalone SDK manager” in the SDK Platforms tab. Update packages by installing all selected ones (already selected). Be sure to install the Android Support Repository under extras. Also while here, install the Google USB Driver under extras for step 7.
6. Go to tools > Android > Android SDK Manager. Click on the SDK Platforms tab. Install the API level(s) relevant to the device you intend to use in step 7. Our code has been tested on APIs 16+. To install all and save yourself from having to make a choice or check your device’s API version, simply tick all API packages from 16-25 and click apply to install them.
7. We strongly recommend you do not use an emulator since the CPU/ABI needs to be arm64 which is 10x slower than x86, which is what most developers will be accustomed to. Using your own device gets around this. If doing so you may need to download ABD drivers for your device. We recommend downloading the ADB Driver Installer and letting it download and install drivers specifically for your device: http://adbdriver.com/. Refer to the following if problems persist and Android Studio does not detect your connected device:
http://stackoverflow.com/questions/16596877/android-studio-doesnt-see-device?page=1&tab=votes#tab-top
8. Now you should be able to run the application on your device: run ‘app’ to do so.

Source Code Directory Structure


src
androidTest
Contains all tests which require the android emulator
test
Contains unit tests which do not require android emulator
main
java
com.cpen321.circuitsolver
model
Contains model classes such as CircuitElm
ngspice
Contains NgSpice related logic and classes
opencv
Contains opencv related logic and classes
service
Contains key logic for integration across components
ui
Contains all Android Activity classes and UI related logic
util
Contains shared constants and helper classes

Guide to Adding a new Circuit Element

Refer to *Circuit Element Class Diagram* for visual representation of CircuitElm classes. The high level steps of creating a new Circuit Element are as follows:

1. Create a new class that extends the CircuitElm class and implements the appropriate methods.
    a. Add any element specific attributes to the class (for example, ResistorElm has a resistance attribute of type double).
    b. Override the constructSpiceElm method. Look up the SPICE specs for the proper string representation of the circuit element so that it can be passed as input to the NgSPICE engine.
    c. Override the onDraw method should be implemented to draw the specific element
    d. Override getVoltageDiff() and getCurrent() appropriately. In most cases, getVoltageDiff should be simply be a matter of finding the difference in value of the voltages of n1 and n2 (n1 and n2 are attributes of CircuitElm).
    e. Refer to the other circuit element classes for an understanding of the basic pattern for circuit element classes
2. Implement image recognition for your element. At this stage, OpenCV only interprets wires and considers all other detected elements as resistors. Further development is necessary for the app to differentiate different circuit elements from an image. Integration of Tensor flow for component recognition is an option.

Running Tests

We provide both circuit logic and UI tests

How to run tests:

For tests that require an emulator (e.g. UI Tests) run the test you are interested in or all in: app\src\androidTest

For test that don’t require an emulator (e.g. circuit logic tests) run the test you are interested in or all in: app\src\test

The following tests are available

In AndroidTest (Require an emulator)

- NgSpiceTest: Calls Ngspice from within the Android app with a preset input net list. Compares Ngspice’s output to the expected output.
- SpiceInterfacerAndroidTest: Tests the entire flow from a circuit described using our own representation, to a solved version of the circuit.
- EspressoUseCase1-5: These have not yet been implemented, but are planned to simulate each use case and ensure that post conditions in all flows

In test (Doesn’t require an emulator)

- SpiceElmTest:
    - This is not really a test, but a way to spot check what the constructSpiceLine() method outputs for different elements. This line is what is inputted into NgSpice for solving the circuit.
- SpiceInterfacerTest:
    - Verifies some simple calls made through the SpiceInterfacer work as expected.
- AnalyzeCircuitImplTest:
    - Verifies that AnalyzeCircuitImpl correctly identifies nodes in a circuit
- CircuitDefParserTest:
    - Verifies that the CircuitDefParser class correctly parses a list of circuit elements into a Circuit Definition Parser, correctly interprets a Circuit Definition file into a list of circuit elements, and correctly scales element coordinates when parsing

Design Patterns Used

Circuit Elements Factory Pattern

- **Rationale**: Encapsulate the creation of circuit elements with the vision of expanding supported element types in future iterations
- Each type of circuit element should have its own class (aka ResistorElm) **which extends the base class, called CircuitElm.**
- **New instances of CircuitElms can be created using the CircuitElmFactory**, which takes in the element type, its coordinates, and its value to create the appropriate CircuitElm.

NgSpice Singleton

- **Rationale**: Prevent creation of more than one NgSpice engine, as only one is necessary for solving a circuit, and creating more than one will waste memory
- The NgSpice class has a static getInstance method which will only create an instance if one has not been created yet.