Google Summer Of Code 2020

Adding SVG editor to RTE

R Rohit Katlaa

March 2020

About You

Personal Information

Name : R Rohit Katlaa
Email Id : rohitkatlaa@gmail.com
Github Handle : rohitkatlaa
College : International Institute of Information Technology
Program : Integrated M.Tech in Computer Science(5 year course)

Why am I interested in working with Oppia?

I have never used online tutorials before my highschool and when I was exposed to the different teaching methods available on the internet, I was stunned. Especially the ones in which you had to follow a definite path towards a goal. That's exactly what I found in Oppia. When I found out about Oppia's mission i.e to provide high quality education to those who lack access to it. It really motivated me to work with Oppia.

As I started working with Oppia I was happy and felt lucky to find such an active and helpful community. The mentors are really helpful and they helped me improve my technical and my coding skills. Contributing to Oppia taught me a lot about Open Source and Project management. So I would like to take this opportunity to develop something significant for the organisation.

What interests me about this project? Why is it worth doing?

As the saying goes 'A picture can tell a thousand words', diagrams and figures aid in learning and understanding concepts better. Giving creators the ability to draw images for the lessons would allow flexibility and ensure that the concepts of the lessons are conveyed clearly without fading the core ideas. I am also interested in this project because no one likes a book without pictures and it is essential to add the correct images to the lessons. So I believe that the concept of adding a svg editor to the application would help the creators convey the lessons better.

Prior experience

I have been involved with web development for the last two years. I am a full-stack developer and I have worked with Ruby on Rails, MERN stack and meteor.

- ➢ My first fullstack project is a Flask messenger application,Hermes which combines several features of popular messenger applications. It includes a global, group and private chat. It also allows users to follow other users and like the posts.

- ➢ At I-Stem Hackathon 2019, my team created an application,Unifundster that interprets stock market data charts to visually disabled people in a way that they can understand. It also included a sentiment analysis of the company data to give a review on the companies.
- ➢ I have worked at Gooru Labs in IIIT-Bangalore, contributing to the Navigated Learning project which provided self directed and personalised education to the individuals at a large scale.
- ➢ I have also created a MERN application,ManageIt, a college complaint portal which includes a flexible role management that can be modified instantaneously.
- ➢ I have had prior work experience with Cirrich, a startup which aims to make websearch simple,relevant and reliable. My work involved automating a scrapper which fetches data from any generic website.

I have also created a HTML canvas based application called WebPaint which is an online application that can be used to draw and take notes and save them instantaneously. I have also helped in the creation of a canvas based framework which was used in our college hackathon. I strongly believe that these projects along with my experience with Oppia for the past six months should help me solve this problem.

Links to my PR for Oppia

Oppia is my first major Open source Organisation I contributed to. I have been in the release testing team of Oppia since September and I am also part of the LACE team of Oppia which has given me a great insight into how a team works and solves issues. Some of my PRs are:

1. Adding a functionality to test the mailgun functionality of Oppia #8686.
2. Added a functionality to shuffle the order of display of multiple choice questions in the lessons #8738.
3. Added UI for editor navbar options for small screens #7557.
4. Added e2e test for code editor interaction #8639.
5. Adds a notification badge for number of open tasks #7865.

Issue: #5436 Working on randomizing the pretest questions.

Contact info and timezone(s)

- Contact:
  - Email: rohitkatlaa@gmail.com
  - Phone: (+91) 8870414906

  I am fine with any form of communication but I would prefer Email.

- TimeZone: Indian Standard Time **(GMT+5:30)**

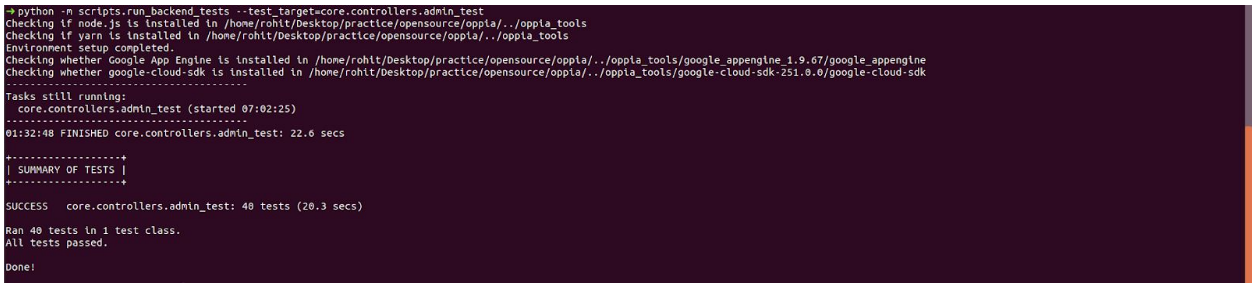
Time commitment

I will be able to commit 4-5 hrs of work per day which can be extended based on the amount of work to be completed. I might have my end semester exams during June(which is not confirmed yet). Other than the exam week I will dedicate the required time for the project.

Essential Prerequisites

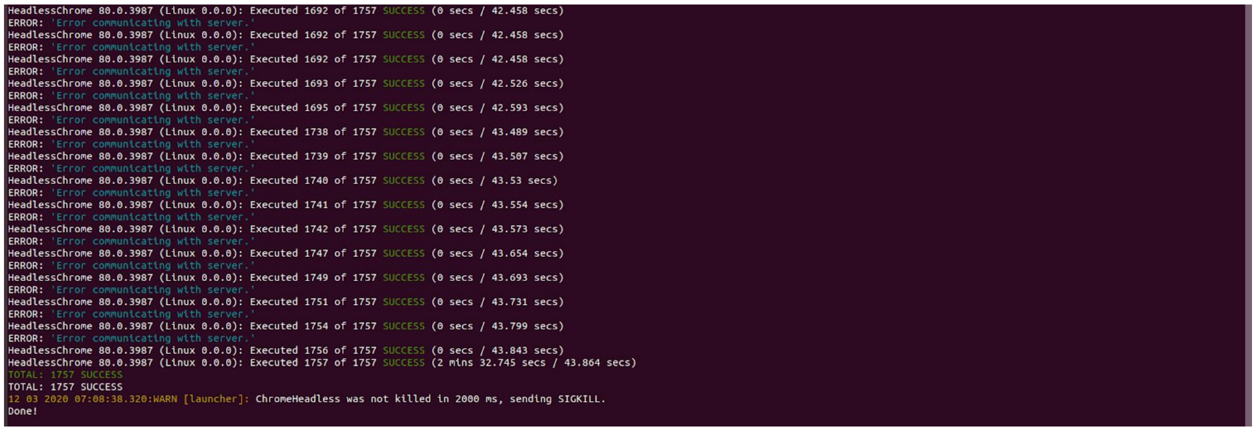
Answer the following questions:

- *I am able to run a single backend test target on my machine. (Show a screenshot of a successful test.)*



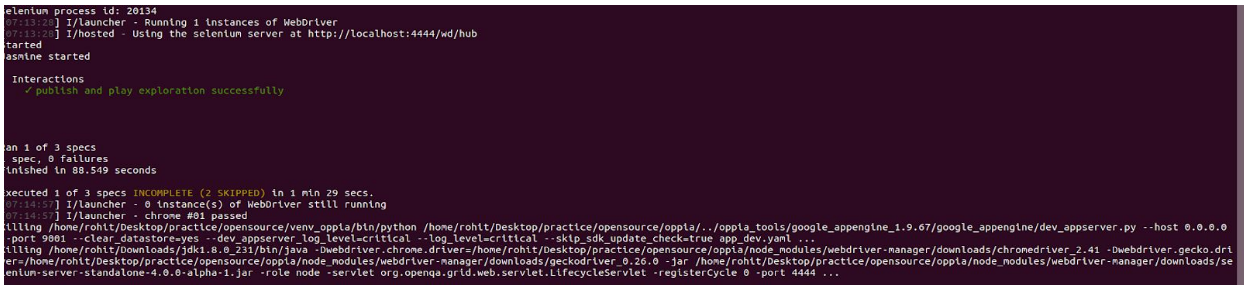
```
➜ python -m scripts.run_backend_tests --test_target=core.controllers.admin_test
Checking if node.js is installed in /home/rohit/Desktop/practice/opensource/oppia/../oppia_tools
Checking if yarn is installed in /home/rohit/Desktop/practice/opensource/oppia/../oppia_tools
Environment setup completed.
Checking whether Google App Engine is installed in /home/rohit/Desktop/practice/opensource/oppia/../oppia_tools/google_appengine_1.9.67/google_appengine
Checking whether google-cloud-sdk is installed in /home/rohit/Desktop/practice/opensource/oppia/../oppia_tools/google-cloud-sdk-251.0.0/google-cloud-sdk
----------------------------------------
Tasks still running:
  core.controllers.admin_test (started 07:02:25)
----------------------------------------
01:32:48 FINISHED core.controllers.admin_test: 22.6 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   core.controllers.admin_test: 40 tests (20.3 secs)

Ran 40 tests in 1 test class.
All tests passed.

Done!
```

- *I am able to run all the frontend tests at once on my machine. (Show a screenshot of a successful test.)*



```
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1692 of 1757 SUCCESS (0 secs / 42.458 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1692 of 1757 SUCCESS (0 secs / 42.458 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1692 of 1757 SUCCESS (0 secs / 42.458 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1693 of 1757 SUCCESS (0 secs / 42.526 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1695 of 1757 SUCCESS (0 secs / 42.593 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1738 of 1757 SUCCESS (0 secs / 43.489 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1739 of 1757 SUCCESS (0 secs / 43.507 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1740 of 1757 SUCCESS (0 secs / 43.53 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1741 of 1757 SUCCESS (0 secs / 43.554 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1742 of 1757 SUCCESS (0 secs / 43.573 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1747 of 1757 SUCCESS (0 secs / 43.654 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1749 of 1757 SUCCESS (0 secs / 43.693 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1751 of 1757 SUCCESS (0 secs / 43.731 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1754 of 1757 SUCCESS (0 secs / 43.799 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1756 of 1757 SUCCESS (0 secs / 43.843 secs)
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1757 of 1757 SUCCESS (2 mins 32.745 secs / 43.864 secs)
TOTAL: 1757 SUCCESS
TOTAL: 1757 SUCCESS
12 03 2020 07:08:38.320:WARN [launcher]: ChromeHeadless was not killed in 2000 ms, sending SIGKILL.
Done!
```

- *I am able to run one suite of e2e tests on my machine. (Show a screenshot of a successful test.)*



```
elenium process id: 20134
07:13:28] I/launcher - Running 1 instances of WebDriver
07:13:28] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
tarted
asmine started

 Interactions
   ✓ publish and play exploration successfully



an 1 of 3 specs
 spec, 0 failures
inished in 88.549 seconds

xecuted 1 of 3 specs INCOMPLETE (2 SKIPPED) in 1 min 29 secs.
07:14:57] I/launcher - 0 instance(s) of WebDriver still running
07:14:57] I/launcher - chrome #01 passed
illing /home/rohit/Desktop/practice/opensource/venv_oppia/bin/python /home/rohit/Desktop/practice/opensource/oppia/../oppia_tools/google_appengine_1.9.67/google_appengine/dev_appserver.py --host 0.0.0.0
-port 9001 --clear_datastore=yes --dev_appserver_log_level=critical --log_level=critical --skip_sdk_update_check=true app_dev.yaml ...
illing /home/rohit/Downloads/jdk1.8.0_231/bin/java -Dwebdriver.chrome.driver=/home/rohit/Desktop/practice/opensource/oppia/node_modules/webdriver-manager/downloads/chromedriver_2.41 -Dwebdriver.gecko.dri
er=/home/rohit/Desktop/practice/opensource/oppia/node_modules/webdriver-manager/downloads/geckodriver_0.26.0 -jar /home/rohit/Desktop/practice/opensource/oppia/node_modules/webdriver-manager/downloads/se
enium-server-standalone-4.0.0-alpha-1.jar -role node -servlet org.openqa.grid.web.servlet.LifecycleServlet -registerCycle 0 -port 4444 ...
```

Other summer obligations

I might have my end semester exams during June(which is not confirmed yet) and my college reopens in August. Other than that I do not have any other commitments. I can assure you that I will keep up with my commitments and complete my targets on time.

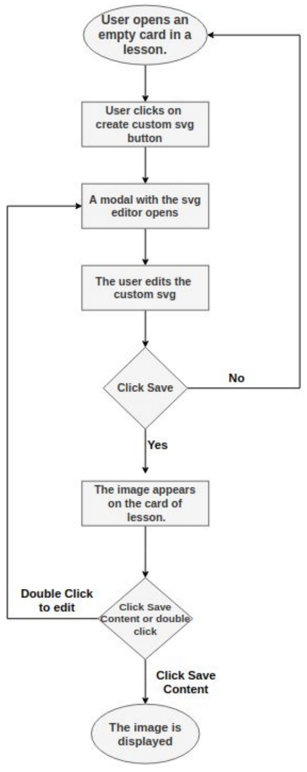
Communication channels

I will be able to communicate with the mentors almost everyday to give my updates on the project. I plan on communicating through Hangouts and mail.

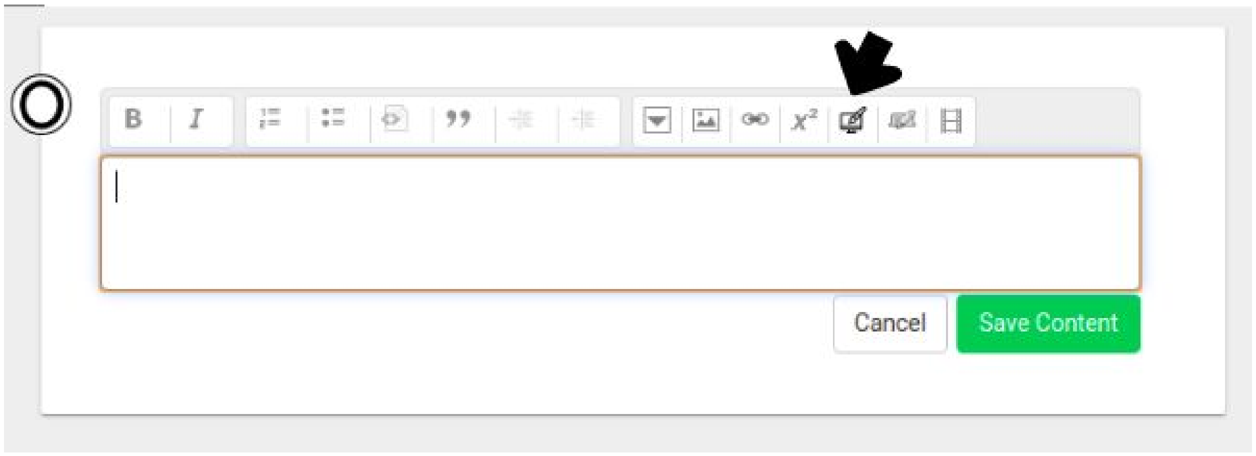
Project Details

Product Design

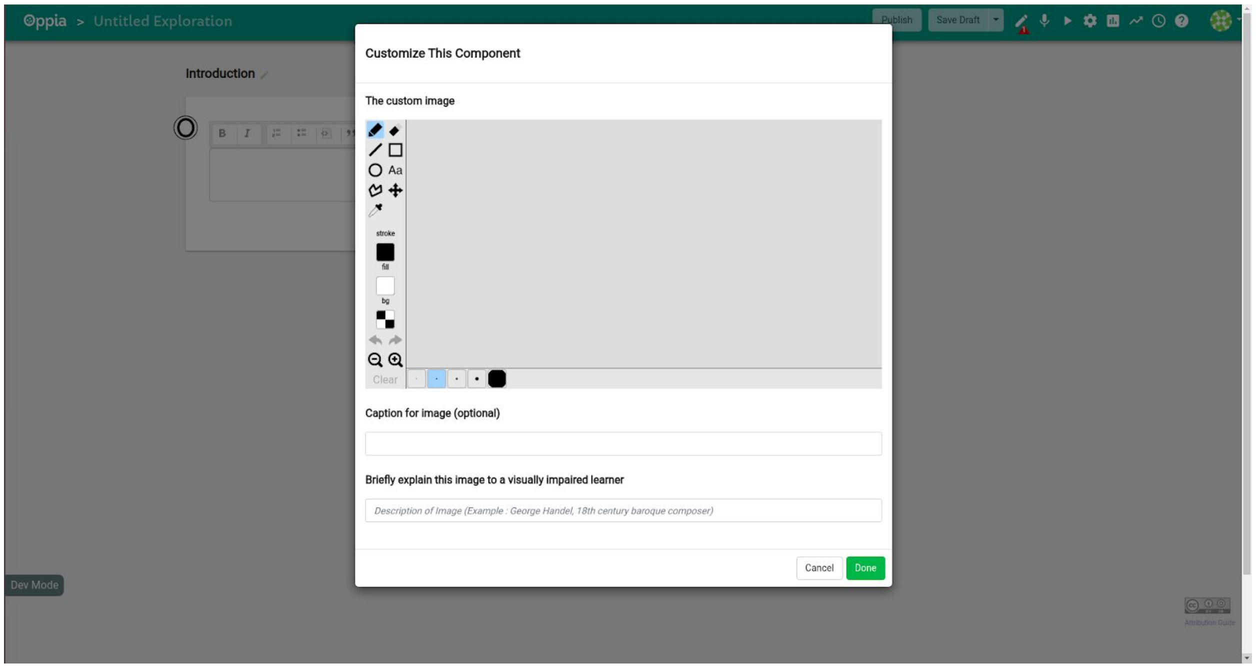
This project is to help the creator of the lessons and artists by providing them with a svg editor in the browser. This can be used to create and edit customized diagrams easily which can be used in the lessons.
Users control flow for creating a custom diagram using the editor is described below.


User opens an empty card in a lesson.
User clicks on create custom svg button
A modal with the svg editor opens
The user edits the custom svg
Click Save
No
Yes
The image appears on the card of lesson.
Double Click to edit
Click Save Content or double click
Click Save Content
The image is displayed

1. Initially the user opens the empty card(or existing) in a lesson.
2. Clicks the create custom svg button among the given options.



3. A modal(a modal is a small window) with the svg editor opens.



Customize This Component

The custom image

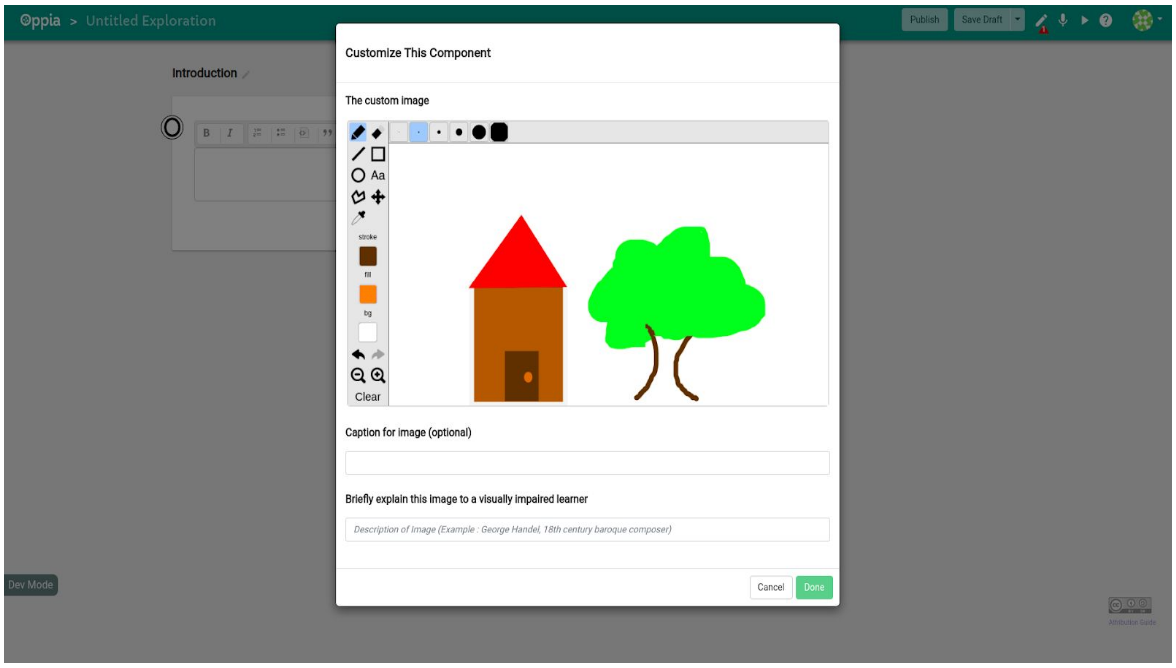
Caption for image (optional)

Briefly explain this image to a visually impaired learner

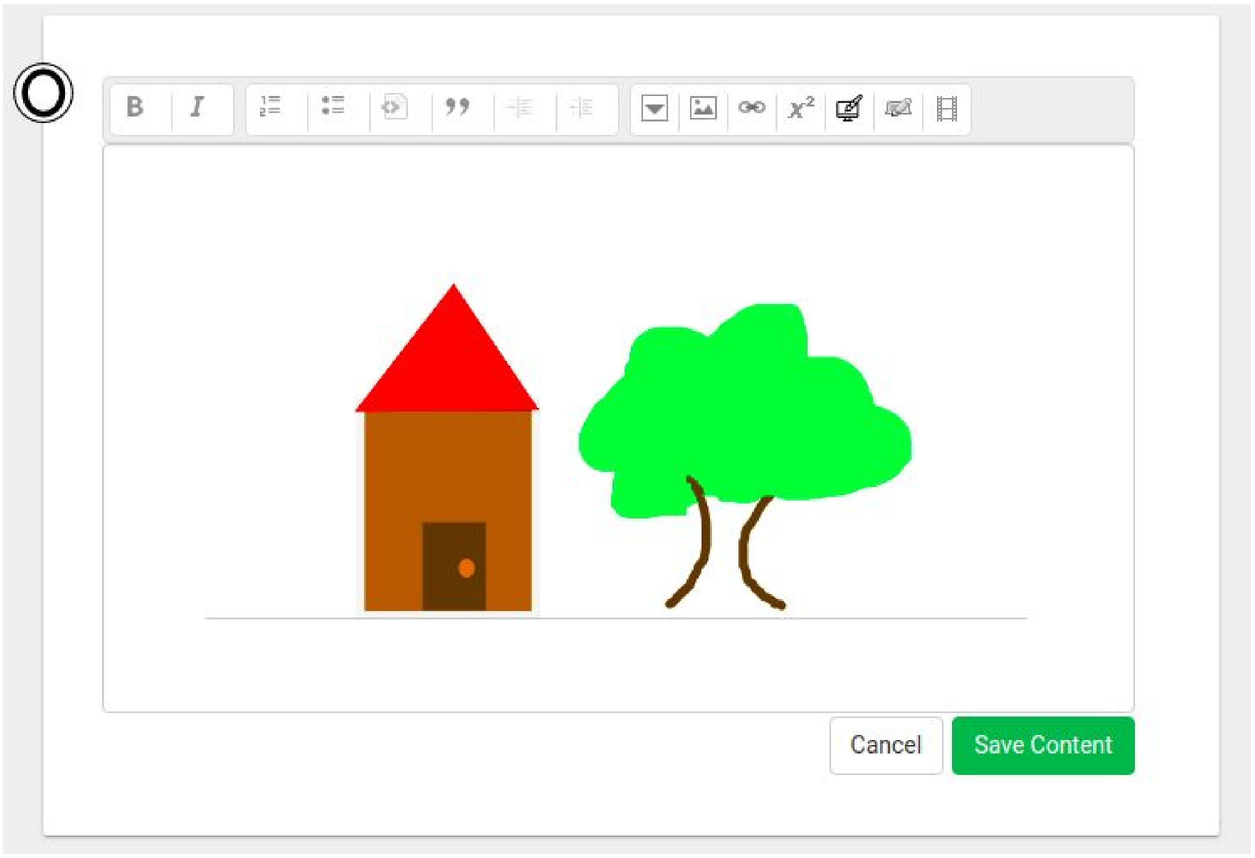
Description of Image (Example : George Handel, 18th century baroque composer)

Cancel Done

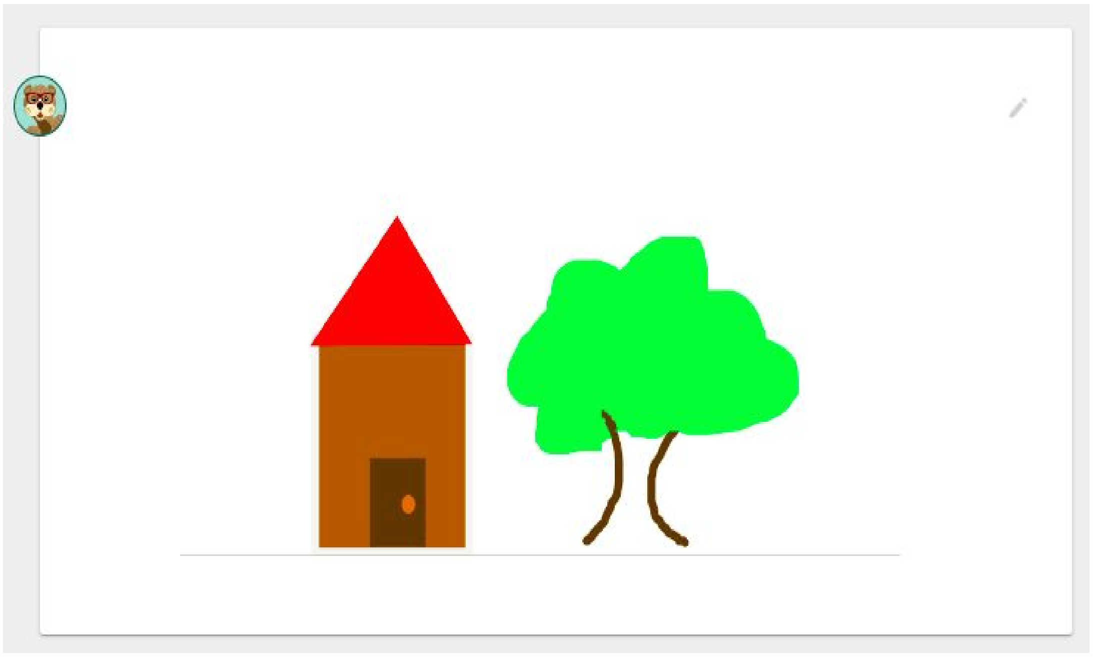
4. The user creates the custom diagram that he/she wants to create using the editor's tools like brush tool, paint tool, polygon tool etc, the caption for the image and the explanation for the visually impaired fields can also be filled. The user then clicks the Done button to proceed and the modal closes.


Oppia > Untitled Exploration
Publish
Save Draft
Introduction
Customize This Component
The custom image
stroke
fill
bg
Clear
Caption for image (optional)
Briefly explain this image to a visually impaired learner
Description of Image (Example : George Handel, 18th century baroque composer)
Cancel
Done
Dev Mode

5. Once the diagram appears the user can click the save content button to save the image to the card or double click on the diagram to edit the svg.



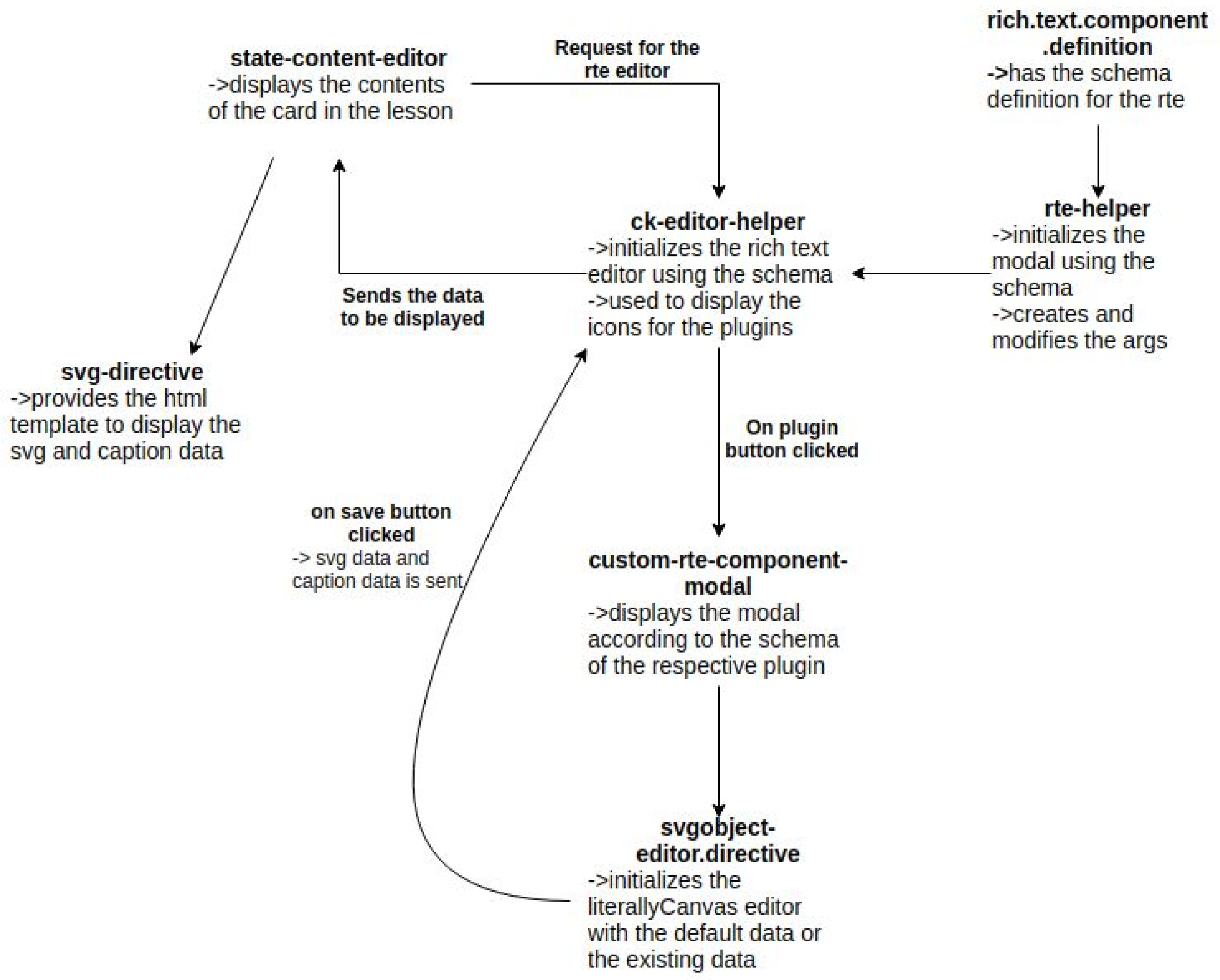
6. Once the image is saved the final output is displayed.



Technical Design

Architectural Overview

The following flowchart explains the frontend flow of the Rich Text Editor with the svg editor.


state-content-editor
->displays the contents of the card in the lesson
Request for the rte editor
rich.text.component .definition
->has the schema definition for the rte
ck-editor-helper
->initializes the rich text editor using the schema
->used to display the icons for the plugins
rte-helper
->initializes the modal using the schema
->creates and modifies the args
Sends the data to be displayed
svg-directive
->provides the html template to display the svg and caption data
On plugin button clicked
on save button clicked
-> svg data and caption data is sent
custom-rte-component-modal
->displays the modal according to the schema of the respective plugin
svgobject-editor.directive
->initializes the literallyCanvas editor with the default data or the existing data

1. The state-content-editor displays the contents of the card and also displays the initial content editor i.e when the card is clicked it opens the RTE editor where all the plugins are displayed.
2. The RTE is initialized by the ck-editor-helper using the schema provided by the rich.text.component.definition. This displays all the plugins using the appropriate icons.
3. When the add diagram(i.e the svg editor) plugin icon is clicked the modal for the plugin is displayed. It contains the LiterallyCanvas Editor and the form for caption data. The template for this modal is in the custom-rte-component-modal file.
4. The Literally Canvas editor is initialized in the svgobject-editor.directive file which is used to create the custom diagram.
5. The svg data and the caption data from the modal is collected when the save button is clicked.
6. This data is then used to display it in the card using svg-directive.html as a template.

Implementation Approach

I will be implementing the editor using LiterallyCanvas.

Why LiterallyCanvas:

I decided to use literally canvas because it is an extensible, open source, HTML5 drawing widget. It also comes with an easy-to-use jQuery plugin.

- LiterallyCanvas is lightweight and it loads much faster than other heavy editors.
- From the user point of view I would prefer it over other editors mainly because literally canvas is extremely smooth in touch devices i.e it didn't cause any problems and it was easy to use whereas SvgEdit created problems with touch devices i.e some of the buttons where not responsive.

  I have tested LiterallyCanvas and SvgEdit in the following devices:
    - Mobile browser
    - Mobile using a stylus
    - Tablet
    - Touch screen laptop
    - And a digital pad that is used to write and teach students

  SvgEdit's buttons were not responsive in the above touch devices. It worked well only on a laptop where a mouse is used. At the same time LiterallyCanvas worked perfectly without any problem.
- LiterallyCanvas is easy to extend because there is support for adding custom tools which can be used to add additional tools like the triangle tool, paint spray tool etc.
- Some of the existing tools in LiterallyCanvas are:
  - Pencil(Brush) Tool
  - Line Tool
  - Circle Tool
  - Text Tool
  - EyeDropper Tool
  - Square Tool
  - Polygon tool
- I plan on introducing this library in Oppia by adding the dependency files in the third party/static folder and importing them wherever is required and by adding the required images in the assets folder.(I have tested this and it seems to work fine without any problem)
  - React is the only main dependency of literallyCanvas. In all, literallyCanvas requires two js files(899.6kB).
    - Literallycanvas.js (192.2kB)
    - React-0.14.3.js (707.5kB)
  - LiterallyCanvas also requires a stylesheet - literallycanvas.css and 26 images(307kB) for the GUI.

Icon to be used for the svg editor:

svg is not as well known, so I thought that a simple icon which clearly denotes a diagram editor, should be used along with a tooltip stating **Insert Diagram**. So I have considered the following icons:

- - a monitor with a paint brush in it
- - a few painting tools on a cloud.
- - another monitor with tools in it.
- - a paint palette.

Any of the above icons should convey the user clearly about the diagram editor. But I would prefer the first icon. This is because it indicates that it is a painting editor and it also indicates that it has touch screen support.

Including the SVG editor:

- Create a folder in extensions/rich_text_components called **svg image** similar to image folder. This folder will contain the folder directives, svg icon that will be displayed in the rte editor and the protractor file that is used for testing(this will be covered in the testing section). The directives file contains the .html file and the .ts file. The html file will contain the template that will be used to display the svg diagram that will be created.

```
<oppia-noninteractive-svg>
      <figure>
<!-- svg tag that is created by the editor -->
         <img ng-src="<[$ctrl.svgUrl]>" alt="<[$ctrl.svgAltText]>">
         <figcaption
style="text-align:center;"><[$ctrl.svgCaption]></figcaption>
      </figure>
</oppia-noninteractive-svg>
```

- Add a schema for the svg editor in the rich_text_components_definitions.ts file. This schema will be used by the rte to create the plugin for the svg editor.
    - Customization_args_specs:
        - Name: Caption

```
{
     "name": "caption",
     "description": "Caption for image (optional)",
     "schema": {
       "type": "unicode"
```

```
      },
      "default_value": ""
    }
```

- Name: svgObject

```
{
      "name": "svgObject",
      "description": "An svg tag which is generated from
the editor",
      "schema": {
        "type": "custom",
        "obj_type": "svgObject"
      },
      "Default_value": ""
}
```

This svgObject is created by the editor after the user created a custom diagram on the svg editor.

- Add a file “svgobject-editor.directive.html” in extensions/objects/templates. This file will be used to initialize the literallyCanvas editor and will be displayed in the modal when the add diagram button is clicked. This file needs to be imported in objectComponentsRequires.ts.
    - Initialize the svg editor

```
 <div class="svg-editor-container">
     <div id="lc"></div>
  </div>
```

```
var lc = LC.init(document.getElementById("lc"), {
       imageSize: {width: 500, height: 300},
       imageURLPrefix: 'path of the image files used for the tools',
       toolbarPosition: 'bottom',
       defaultStrokeWidth: 2,
       strokeWidths: [1, 2, 3, 5, 30],
     });
```

    - Saving the image and generating the Svg Tag for displaying in the card.
        - A simple sanity check to check if the svg string is empty or not and also to check if the editor is empty.

        - There shouldn’t be any other threats from svg because XSS attacks are avoided if the svg is used as an image source in the img tag.
        - The svg tag that is outputted by the getSVGString is not formatted correctly. It has to be formatted correctly so that it can be rendered again into the LiterallyCanvas editor. I have identified the mistake, the svg tag generated has some problem with the spacing after the double quotes. This does not cause a problem when we are trying to display it but it causes a problem when I am trying to convert it into dataUrl which will be used to upload into the editor for editing. I will be correcting this mistake before sending it to the backend.

```
var svgData = lc.getSVGString()
ctrl.localValue = svgData
```

    - Once the svg data is generated the svg data can be sent to the backend to save it as an svg file using the existing service(ImageUploadHandler)
        - Fire a POST request with the svg data

```
ctrl.entityId = ContextService.getEntityId();
ctrl.entityType = ContextService.getEntityType();
let form = new FormData();
form.append('image', svgData);
form.append('payload', JSON.stringify({
  filename: ImageUploadHelperService.generateImageFilename(
    dimensions.height, dimensions.width, 'svg')
}));
var imageUploadUrlTemplate = '/createhandler/imageupload/' +
  '<entity_type>/<entity_id>';
CsrfTokenService.getTokenAsync().then(function(token) {
  form.append('csrf_token', token);
  $.ajax({
    url: UrlInterpolationService.interpolateUrl(
      imageUploadUrlTemplate, {
        entity_type: ctrl.entityType,
        entity_id: ctrl.entityId
      }
    ),
    data: form,
    processData: false,
    contentType: false,
    type: 'POST',
    dataType: 'text'
```

```
           })
```

        - Change the feconf to accepts svg extension
        - Note that **imghdr.what()** in editor.py does not recognize the svg data so svg has to be identified
        - Also note that in fs_services.py the compress image is not necessary for svg

    - Fetching the file from the backend
        - The source of the file can be obtained from:

```
var getTrustedResourceUrlForImageFileName = function(imageFileName) {
         var encodedFilepath = window.encodeURIComponent(imageFileName);
         return $sce.trustAsResourceUrl(
           AssetsBackendApiService.getImageUrlForPreview(
             ctrl.entityType, ctrl.entityId, encodedFilepath));
       };
svgUrl = getTrustedResourceUrlForImageFileName(filename);
```

This svgUrl can be used as a source in the img tag.

    - Loading the image into the literallyCanvas editor:

```
      lc.saveShape(LC.createShape('Image', {x: 0, y: 0, image:
imageSrc}));
```

The above code will upload the existing image and this can be edited in the editor.

E2E Testing:

- Create a new file called protractor.js in extensions/rich_text_components/ svg_image as mentioned above.
- Import this file in extensions/rich_text_components/protractor.js file which can be then used in extension.js through forms.js
- The testing for the editor can be done by drawing a line with the default brush tool and then checking if the generated svg is the same as the actual svg.
    - The line can be drawn using the browser.actions() function.

```
browser.actions().
   mouseDown(svgEditorContainer, {x: xOffset1, y: yOffset1}).
   mouseMove(svgEditorContainer, {x: xOffset1, y: yOffset1}).
   mouseUp().
perform();
```

  - Then the save button can be clicked which will generate the svg data.
- The expectComponentDetailsToMatch function will check whether the editor is present and it also checks if the generated svg data is correct.

Adding to existing validations

Html_validation_service.py consists of some validations for the contents that will be displayed. It ensures that the HTML code is compatible with RTE and CKEditor. But since the html data sent to the backend resembles closely to the image plugin of the RTE not many changes are required. Some of the changes are:

- Add oppia-noninteractive-svg to feconf.RTE_CONTENT_SPEC
  - RTE_TYPE_TEXTANGULAR

```
'oppia-noninteractive-svg': ['b', 'i', 'li', 'p', 'pre'],
```

  - RTE_TYPE_CKEDITOR

```
'oppia-noninteractive-svg': ['blockquote', 'li',
'[document]'],
```

  - Also add it to ALLOWED_TAG_LIST in textangular and ckeditor
- It is also necessary to add the correct extension to feconf.ALLOWED_RTE_EXTENSIONS

Since any html data that is sent to the backend is generated by the editor there is no need for complicated validations of the svg data.

- Add RTE validations to Components.py

```
class SvgImage(BaseRteComponent):
   """Class for Image component."""

   @classmethod
   def validate(cls, value_dict):
       """Validates Image component."""
       super(SvgImage, cls).validate(value_dict)
       filename_re = r'^[A-Za-z0-9+/_-]*\.((svg))$'
       filepath = value_dict['filepath-with-value']
       if not re.match(filename_re, filepath):
           raise Exception('Invalid filepath')
```

- Add the tests to test the validations in components_test.py

```
def test_svg_image_validation(self):
       """Tests svg image component validation."""
       valid_items = [{
           'filepath-with-value': 'random.svg',
           'alt-with-value': '1234',
           'caption-with-value': 'hello'
       }, {
           'filepath-with-value': 'xyz.svg',
           'alt-with-value': 'hello',
           'caption-with-value': 'abc'
       }]
       invalid_items = [{
           'filepath-with-value': 'random.png',
           'caption-with-value': 'abc'
       }, {
           'filepath-with-value': 'xyz.svg.svg',
           'alt-with-value': 'hello',
           'caption-with-value': 'abc'
       }, {
           'filepath-with-value': 'xyz.png.svg',
           'alt-with-value': 'hello',
           'caption-with-value': 'abc'
       }]

       self.check_validation(
           components.Image, valid_items, invalid_items)
```

Add custom tools

General directory structure:

Create a directory for literallyCanvas, the main html and ts file in it. A sub directory for custom tools. It consists of only js files. Each js file for each tool.

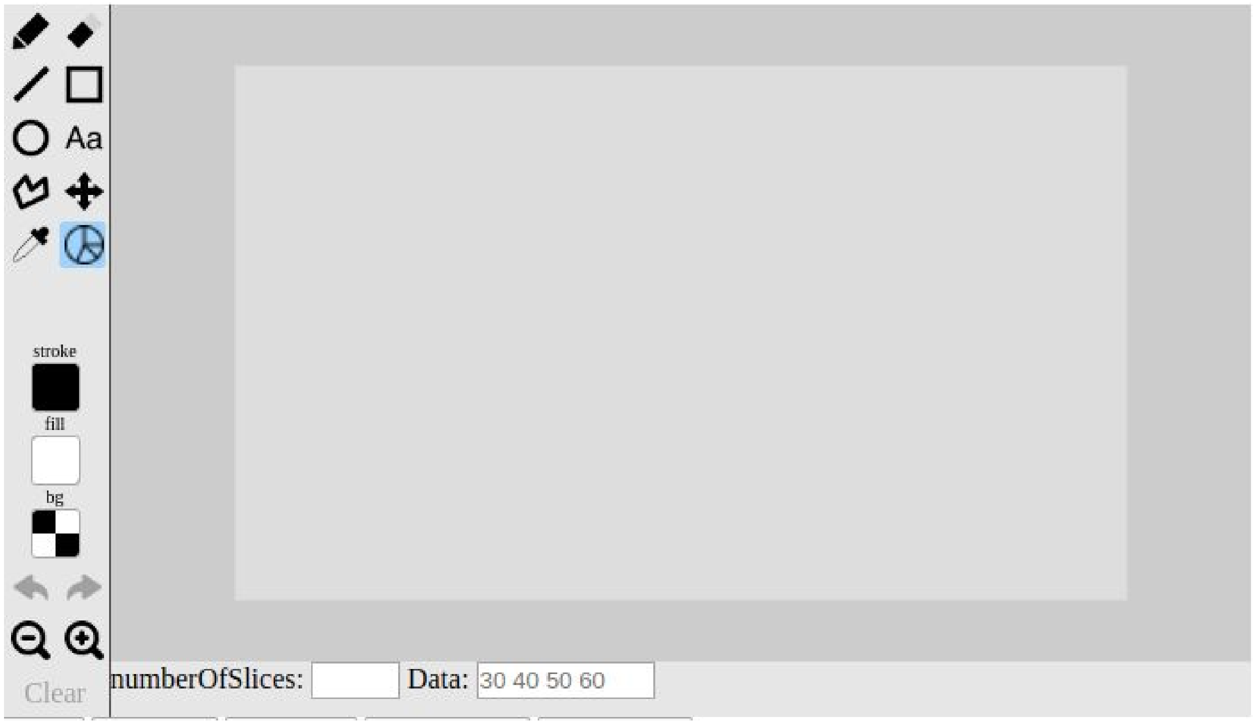
I am planning on using this API for creating the tools.

Tool1: Pie chart tool

Icon:

This tool can be used to draw pie charts in the required proportions.
This uses two inputs: number of parts and value of each part. Given these inputs the required pie chart can be drawn.

- First get the inputs from the user.



numberOfSlices: Data: 30 40 50 60

The below two inputs can be used to get the required data.

- The numberOfSlices gives the required number of parts in the pie diagram.
- The spaced data gives the values of each part of the pie chart.

```
function drawLine(ctx, startX, startY, endX, endY){
   ctx.beginPath();
   ctx.moveTo(startX,startY);
   ctx.lineTo(endX,endY);
   ctx.stroke();
}
function drawArc(ctx, centerX, centerY, radius, startAngle, endAngle){
   ctx.beginPath();
   ctx.arc(centerX, centerY, radius, startAngle, endAngle);
   ctx.stroke();
}
function drawPieSlice(ctx,centerX, centerY, radius, startAngle, endAngle,
color ){
   ctx.fillStyle = color;
   ctx.beginPath();
   ctx.moveTo(centerX,centerY);
   ctx.arc(centerX, centerY, radius, startAngle, endAngle);
   ctx.closePath();
   ctx.fill();
```

```
}

var Piechart = function(options){
   this.options = options;
   this.canvas = options.canvas;
   this.ctx = this.canvas.getContext("2d");
   this.colors = options.colors;
   this.draw = function(){
       var total_value = 0;
       var color_index = 0;
       for (var categ in this.options.data){
           var val = this.options.data[categ];
           total_value += val;
       }
       var start_angle = 0;
       for (categ in this.options.data){
           val = this.options.data[categ];
           var slice_angle = 2 * Math.PI * val / total_value;
           drawPieSlice(
               this.ctx,
               this.canvas.width/2,
               this.canvas.height/2,
               Math.min(this.canvas.width/2,this.canvas.height/2),
               start_angle,
               start_angle+slice_angle,
               this.colors[color_index%this.colors.length]
           );
           start_angle += slice_angle;
           color_index++;
       }
   }
}

var DrawPieChart = function(newCanvas, newCtx, numberOfParts, values){
   data = values.split(" ");
   for(var i in data) {
       data[i] = parseInt(data[i]);
```

```
        if(Number.isNaN(data[i])){
            return "Invalid data";
        }
    }
    if(numberOfParts != data.length){
        return "Invalid Data"
    }
    var myPiechart = new Piechart(
        {
            canvas:newCanvas,
            data: data,
            colors:["#fde23e","#f16e23", "#57d9ff","#937e88"]
        }
    );
    myPiechart.draw();
}
```

The above code implements the drawPieChart function which can be used to draw the pie chart for the intended data on the canvas element sent to the function.

- Draw the pie chart in a temporary canvas.

```
      var newCanvas = document.createElement('canvas');
      newCanvas.backgroundColor = 'transparent';
      var newCtx = newCanvas.getContext('2d');
      drawPieChart(newCanvas, newCtx, numberOfParts, values)
```

- Get the dataUrl from the canvas

```
      var myimage = new Image();
      myimage.src = newCanvas.toDataURL();
```

- Create the shape in the editor.

```
      LC.createShape('Image',{x:pt.x,y:pt.y,image:myimage})
```

In Short: The tool will draw the pie chart on a canvas and get the image src. This image will be displayed in the literallyCanvas editor.

Why is this tool required: This tool helps to create simple pie charts which can be used for various mathematical applications like displaying fractions, statistics etc.

Tool2: Arc Tool

Icon:

- An arc can be generated using three points(center,start,stop). The start and stop gives the angle with respect to the center and this can be used to generate the required arc.

```
    lc.canvas.addEventListener("click", myOnClick, false);
    function myOnClick(e) {
           var element = lc.canvas;
           var offsetX = 0, offsetY = 0

               if (element.offsetParent) {
             do {
               offsetX += element.offsetLeft;
               offsetY += element.offsetTop;
             } while ((element = element.offsetParent));
           }
           var x = e.pageX - offsetX;
           var y = e.pageY - offsetY;
         }
```

- Thus the above function returns the x and y coord which can be used to determine the center, start and stop points.
- Then the set of points along the path can be calculated and then the points can be joined using the linepath function with a smooth curve.

```
     lc.saveShape(LC.createShape('LinePath',{points: ArcPoints}))
```

Why is this tool required: This tool helps to create arcs and arc is a basic component of geometry.

Tool 3: Import Image tool

Icon:

This tool can be used to import images into the literallyCanvas editor.

- First upload the image to get the dataURL.

```
    var file     = document.getElementById('img-upload').files[0];
     var reader   = new FileReader();
     reader.addEventListener("load", function () {
       var img = new Image();
```

```
        img.src = reader.result;
      }, false);

      if (file) {
        reader.readAsDataURL(file);
      }
```

- Once the image’s dataUrl is generated, the scale of the image can be selected from the custom styles.
- The position of placement can be determined by allowing the user to hover and place the image anywhere on the canvas
- Then the image can be drawn on the canvas using the following code.

```
lc.saveShape(LC.createShape('Image', {x: xCoord, y: yCoord, image: img,
scale: selectedScale}));
```

Why is this tool required: This tool is essential because it allows the user to import any custom image into the editor.

Milestones

I plan to have three big milestones based on the timeline provided by Google. Below are the detailed explanations of each milestone.

Preparation / Community Bonding Period (Now - June 1):

During this period I will continue contributing to the Oppia community. I will also try to get more familiar with the codebase and the mentors. If possible I will start working on the project immediately so that I might be able to avoid unforeseen delays in the future.

Milestone 1

Key Objective:
- The required backend changes and validations will be in place.
- It is possible to create and save the diagrams, but the UI will be hidden behind a flag variable until the e2e testing functionality and backend changes which is to be included in the 2nd milestone.
- The saved svg diagrams will also be displayed in the card content.

No.	Description of PR	Target date for PR submission	Target date for PR to be merged
1.1	Add the required Backend changes and required validations	4 - 5 June 2020	6 - 7 June 2020
1.2	Adding the literallyCanvas to Oppia - changing schema and initializing literallyCanvas without save functionality	12 - 13 June 2020	15 -16 June 2020
1.3	Add the saving functionality to the Svg editor	21 - 22 June 2020	24 -25 June 2020
1.4	Fetch and display the saved svg image along with re-editing the image.	30 June - 1 July 2020	2 - 3 July 2020

Milestone 2

Key Objective:

- The E2E test suite for the svg editor will be ready.
- Once the e2e test suite and the required backend changes are added, the UI will be available for the user. So the user will be able to create and save the diagrams. And the saved diagrams will be displayed in the card content of the exploration.
- The user will be able to use the pie chart tool to add pie charts to the editor.
- The last week of July will be considered as a buffer to prevent any delays caused due to my end-semester exams(expected to be in July).

No.	Description of PR	Target date for PR submission	Target date for PR to be merged
2.1	Add E2E testing for the LiterallyCanvas editor	8 - 9 July 2020	11 - 12 July 2020
2.2	Add the pie chart tool to the editor	17 - 18 July 2020	20 - 21 July 2020

Milestone 3

Key Objective:

- The user will be able to use the arc tool to draw arcs.
- The user will be able to use the import image tool to import the required image into the editor.

No.	Description of PR	Target date for PR submission	Target date for PR to be merged
3.1	Add the Arc tool to the editor	9-10 August 2020	12-13 August 2020
3.2	Add the Import image tool to the editor	20-21 August 2020	23-24 August 2020

Future Work

- More custom tools and shapes can be added to the editor. For example arrow tool, cone shape etc.
- E2E testing for the custom tools that are created for the editor.
- Make the editor more accessible to physically disabled people.

Additional Project-Specific Considerations

Documentation Changes

- Adding a section in the RTE wiki page explaining how literallyCanvas is connected and initialized.
- Adding a wiki page explaining how to create custom tools in the editor.