Disney Online Kerpoof Studios

Adam Bodnar

Kelsey Kopecky

Max Mazzocchi

Johnny Zabonik

Advanced Software Engineering

May 13th - June 21st, 2013

Disney Online Kerpoof Studios is a part of Disney concerned with creating online artistic tools and activities for children featuring characters from Disney TV shows or original art. The users of the Disney Create site go online and use tools to create art, animations, or photo mash-ups which can then be posted to the Disney Create social network. From there, other users can rate, comment, or critique the artwork posted, allowing popular creations to be displayed more prominently and help users improve their artistic skills. Currently, the site provides three tools to users: a “digital painter” which allows users to draw with brushes, stamps, and other basic drawing tools; a “photo mash-up” which allows users to add pictures of their favorite Disney characters and create a scene; and a flip-book animation tool which allows users to create basic animations based on their art. Currently Disney Create implements these tools using Flash, which unfortunately limits their market audience and their compatible technology. As of right now, Disney Create is attempting to move more towards a globally compatible program.

The main idea of our project was to take the ideas behind two of the existing flash applications hosted on Disney Create and convert them to use HTML5 and Javascript. This will make them usable on tablets and phones as well as full computers. The app will combine the drawing capabilities of both the digital painter and the photo-mash up. The combined application will enable users to draw, paste in images, use stamps, add backgrounds, and more in a vector format. This vector format will allow translation, rotation, scaling, and re-layering of images and components. In addition all tools must be compatible with keyboard and mouse as well as touch interfaces.

Functional Requirements

Stamps: a set picture that can be replaced and reused multiple times
- Select after stamping
- Scale/Rotate/Reflect

Fills: fill in area bounded by lines or objects
Backgrounds: change background layer
Text Input: allow users to create a “text object” which can include speech bubbles
- Text boxes
- Wrap text

Draw: a basic line drawing tool.
- Brush, pencil, calligraphy, spray can
- Change color
- Change line width
- Change opacity
- Line correction (smooth out rough edges)

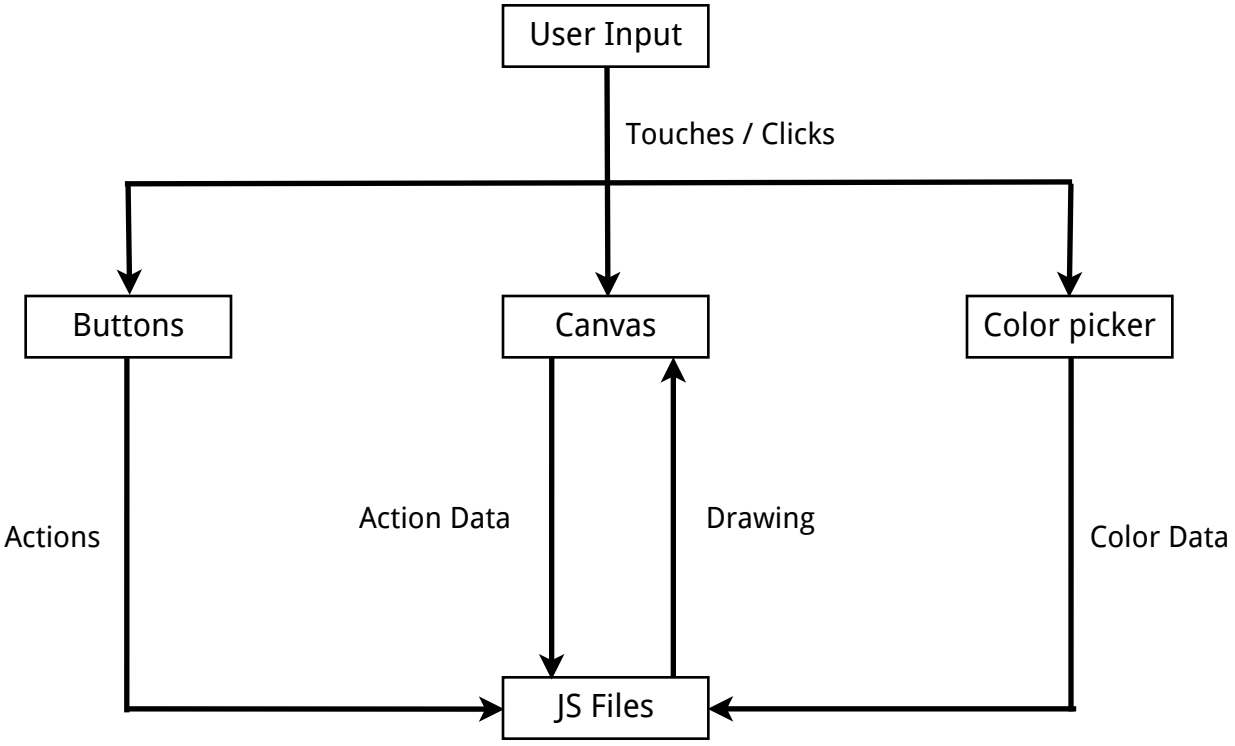
Eraser: erase whole objects or layers by clicking or dragging
Undo/Redo: undo or redo actions including creation or transformation of layers
Select: allow selection and movements of layers and objects
- Select entire objects
- Move selections
- Scale/rotate selections

Save: export to a compressed format which can then be reloaded
- Save/share and open (title art)

Non-Functional Requirements

- Must use HTML5 and Javascript
- Must be compatible with iOS, Android, and other mobile browsers without using platform-specific components
- Must be intuitive enough to be easily used by a demographic of 8-12 year old children
- Must be easy to extend with new images and imported content

System Architecture



Interface

When interactions occur with the application, they come through one of three different avenues: touches (in the case of mobile devices), clicks (in the case of full computers), and keyboard input (used in both mobile and full). In the case of touches and clicks, there are three separate sections that are responsive; this includes the canvas itself, the buttons used to select tools, and the color picker and sliders. In all of these cases, the actions are “captured” by Javascript and jQuery event listeners. The event listeners are bound to the individual components, and call functions contained in kiwilib.js that process the input.

kiwilib.js

kiwilib.js holds the main functions for binding event listeners, processing events, and rendering the canvas. kiwlib.js’ main purpose is to provide a centralized backbone for the rest of the application to build off; it holds the object hash, the layer list, and the action stack. In addition, kiwilib.js has the responsibility of rendering the canvas correctly, including any

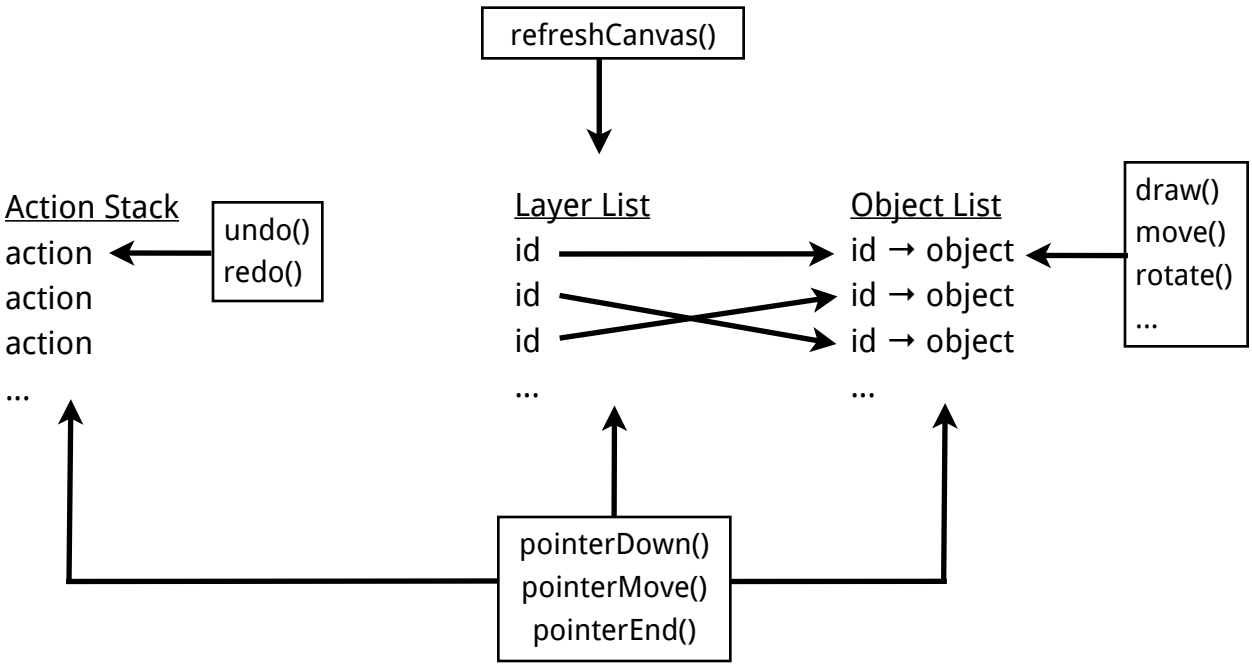
transformations made on the canvas itself, and calling the draw functions of each layer in the correct order.

Object/Utility Files

Each object or utility file provides definitions and methods specific to its purpose. kiwilines.js, for example, provides methods for the construction and transformation of lines; kiwisave.js provides functionality for saving and loading creations via JSON format. The methods contained within each file are called almost exclusively within the file itself or by kiwilib.js; this allows each file to exist independently of the others, relying on kiwilib.js to process interactions between them.

Technical Design

Object, Layer, and Action Interaction



kiwilib.js, as mentioned above, provides the core functionality for processing events and directing the flow of the program. Key to this is the object list, layer list, and action stack, which define the current state of the canvas and program as a whole at any given time.

First of these is the object list; this is represented as a Javascript “object” or dictionary. Every object in the object list is assigned a unique id upon creation; this id is the key that will map to the object in the list. Objects are never stored anywhere else and are never deleted, even if erased; they permanently reside in the object list in case an undo is called. The exception to this is when the canvas is completely cleared; in this case, the entire program is restored to its initial state.

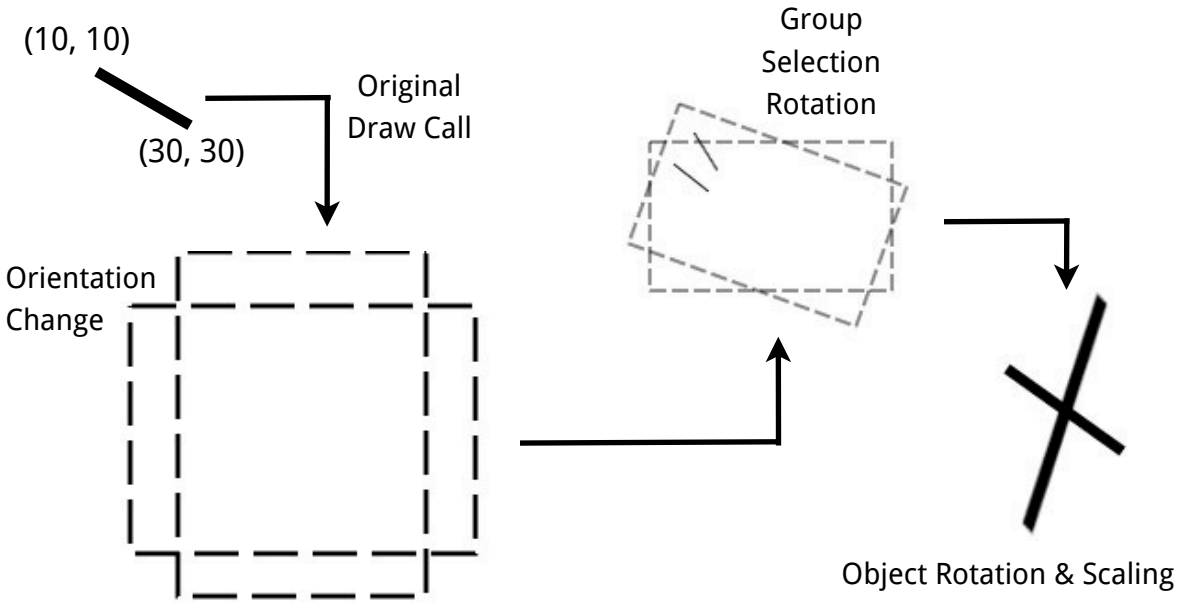
Next is the layer list, which works closely with the object list. The layer list is an array with each cell representing a layer of the drawing. Each layer contains one id, which corresponds to one object in the object list. It is important to note that the layer list does not contain the objects themselves, but only their ids; this allows objects to be edited within the object list without affecting the ordering of the layers. No two objects can exist within the same layer; the exception to this rule is a “bound” object, which represents a group of zero or more objects bound

together. In this case, the layer holds an id which corresponds to the bound object, which in turn contains its own “mini-layer-list” holding the ids of the bound objects. Layers are drawn in ascending order, with layer 0 drawn first on the “bottom” of the canvas, and layer *n* drawn last, on “top” of all the others. When a new object is added, it is placed on the top of the layer list, and can be moved up or down the list using layer controls.

Finally, the action stack keeps track of all actions responsive to the “undo” and “redo” commands. These actions include object creation, translation, rotation, and scaling; erasing; layer editing; background editing; and text modification. When one of these actions is performed, an “action object” is generated. The action object is an object with only two functions: undo() and redo(). Each of these can be called to undo or redo the action they correspond to. Interestingly, saving an action on the stack is identical to saving the state of the canvas at that point in time; for example, say object #5 is created and added to the top layer of the layer list. At the same time, action #5 is generated, in which the undo function simply removes the top layer of the layer list. Ordinarily, this would be a dangerous function, as the top layer of layer list is not always object #5. However, if the layer list is edited in any way, including swapping layers or adding more objects, new actions are created and placed “above” action #5; action #5 cannot be undone until the actions above it are undone. Therefore, in order to access action #5, the canvas must be restored to exactly the same state as it was when action #5 was created, making the undo function completely safe.

As input is is received, kiwilib.js calls pointerStart(), pointerMove(), and pointerEnd() accordingly. By interpreting movements of the user, along with the current tool, objects can be created and transformed. When an object is created, it is initialized with basic information by kiwilib.js, and assigned an id. The id is then placed at the topmost layer of the layer list, ensuring it will be drawn last. The reference to the object is then passed to a specific utility file, which creates the object’s specific functions. Each object, regardless of its type, must contain a set of functions called by kiwilib.js; these include draw(), which draws the particular object; select(), which, given a set of coordinates, returns whether it is inside this object’s bounds; rotate(), scale(), move(), and more. When kiwilib.js interacts with the object, it will use these methods.

Drawing the Canvas



Drawing individual objects on the canvas is a challenge because there are numerous factors affecting where and how an object is actually drawn. For example, a line segment could extend from point (10, 10) to point (50, 50). Drawing it could simply be a drawLine() call with the intended coordinates; however, the line is rotated 35 degrees around its midpoint and scaled by a factor of 1.3 in the y direction. In addition, it is part of a group object that has been rotated around its own midpoint (which is different than the line's midpoint) -42 degrees and scaled by a factor of 0.7. This is after the canvas itself has been translated in the positive y direction due to the menu bar, and the entire device has been rotated 90 degrees to the left in order to view the drawing in landscape. These translations could affect anything from our simple 2-point line segment all the way up to a flood fill object containing 15 sectors, with 30 points each. Instead of performing complex mathematical transformations on each point, a simpler solution was found that allows every transformation to be made quickly and effectively.

All drawing on an HTML canvas goes through a specific Javascript component of the canvas called the "context". A canvas' context can be accessed at any point. Drawing is simply a call like context.drawSomething(coordinates). The context itself has the ability to be transformed through translation, rotation, and scaling; a context that is translated 30 pixels to the right will draw everything afterwards 30 pixels to the right. This is important, because if our two-point-line

is scaled by 1.3 in the y direction, we simply scale the whole canvas and draw the line as usual.

This simplifies drawing tremendously; again, we can look at the example of the two-point-line. As the draw function gets called, the context is rotated to accommodate the orientation of the device. It is then translated for the menu bar, rotated/scaled about the midpoint of the group object, and finally rotated/scaled about the midpoint of the line. The line is then drawn with its original coordinates: (10, 10) and (50, 50), oblivious to the transformations that came before it, but still appearing in the right place on the screen.

This method of drawing does introduce some minor issues. When scaling or rotating the context, the context will always scale and rotate about the origin: (0, 0). This is a problem, as objects intuitively rotate and scale about their midpoint, which can be anywhere on the canvas. To accommodate this, a new way of drawing objects was produced. First, objects are always drawn centered at (0, 0). This was a simple calculation to make; for our segment, the midpoint would be (30, 30), so 30 is subtracted from every x and y coordinate, making the new points of the segment (-20, -20), and (20, 20). This is obviously centered at the origin. Next, the context itself is translated to the midpoint of the object; in our case, the context is translated 30 in the x direction and 30 in the y direction. This process seems meaningless; the line segment is now draw in the exact place it used to be. However, the midpoint of the segment is now (0, 0), so when the context is rotated and scaled, it transforms about the midpoint of the segment. Repeating this process for every object allows transformations to occur about the midpoints of the objects without any complex math.

Another issue presented by this method of drawing was the issue of a universal context. The context is being handed off to each individual object as it is being drawn; because the changes to the context are persistent, the context that is handed back from the draw call is transformed completely from the context before. Rather than save every transformation and reverse it afterwards, a save and restore system was used. Context contains a built-in save() method and restore() method; the save() method saves the exact state of the of the context including any transformations made up to this point. The context is then rotated and scaled multiple times, distorting it from its original configuration. Before it is handed back, restore() is called, which rolls back the context to the last save point. By doing this, the original configuration can be preserved throughout every draw call.

Design Decisions

- JavaScript and HTML5 Canvas are able to be viewed on any modern browser along with tablets and phones. Flash (which was what was being used before) cannot be used on mobile devices and has some version issues on desktops.
- JQuery helps making the code more readable and the application itself easier to use. Its binding functions allow an interface to be built quickly and efficiently.
- Vector Graphics (instead of pixel graphics) gave us scalability and manipulation without losing any details. While this does required the use of SVGs, we do believe the benefits outweigh the costs.
- Object Oriented coding makes lines, shapes, etc. easier to represent. The code also has a centralized structure containing a list of all layers and a hash of all objects. Although JS isn't an OO language primarily, we can still use OO principles to interface more easily with other components of the project.
- A Static Canvas saves the aspect ratio of the canvas on mobile devices when the orientation is changed. The automatic rotation of the canvas in a mobile device causes the artwork to be covered or lost; the automatic reorientation keeps the canvas oriented to the device.
- Rejected the VM that Disney uses due to performance problems on our computers. We are using our local operating systems because we are more comfortable with them and they perform better.
- Cached SVGs/Drawings (CanVG) allow for faster rendering and performance. Rendering all SVGs every time a draw call is made severely effected performance. In addition caching the canvas as a whole requires only new or changing layers to be drawn.
- GitHub works well for source control.
- MongoDB is consistent with other Disney products. Although our MongoDB interface hasn't be built yet, it will later be integrated into their site.

Results

Our goal was to create a digital painter and photo mash-up tool in HTML5 and JS in order to create a desktop and mobile compatible web application. At the end of our project, we achieved full functionality in Google Chrome, IE, and majority of mobile browsers. In addition, we have tested on both Android and Kindle tablets, as well as phones. We completed basic drawing tools, transformation of objects, layer controls, undo/redo, and various other drawing features. Specifically: pen tool, including brush, pencil, calligraphy, and spray can; fill tool using both patterns and solid colors; shapes, including line, circle rectangle, and triangle; stamps; text boxes and callout bubbles; selection tool; erase tool; color picker used to define color, tint, thickness, and opacity; eyedropper tool; zoom tool; undo, redo; clear; copy and paste. One of the requirements we did not complete was the save and load component. A drawing can successfully be converted into a JSON string, and then re-created at a later date using that same string. Unfortunately, the JSON string cannot be stored anywhere, as we did not have access to the DIsney database. One of the biggest non-technical challenges has been the configuration of our workspace; one of the biggest technical challenges has been that HTML5 is still very new and under development. This project has taught us significant amounts about web design, graphical applications of mathematical formulas, and the capabilities of the canvas tag. Overall, we delivered a successful and functional product that met our clients specifications.

Appendices

Installation

This is a web based application, so a web server is required. The specific requirements for the server are listed below:

- MongoDB
- PHP, Ruby, or another server side scripting language
- Ajax

index.html is the core page for the program. It relies on the css, js, and img directories.

Development Environment

Development was performed on personal computers and on multiple operating systems, including Windows 7, Mac OSX Lion, Linux Mint; in addition, multiple browsers were tested, including Firefox, Google Chrome, IE, Safari, Chrome, Amazon Silk, and other mobile browsers. A common repository was kept on GitHub and a web implementation was tested on the CSM Illuminate servers.

Coding Conventions

We used primarily an object oriented approach. This allowed us to define individual rendering, selection, and transformation methods for each type of object in addition to individual instances of that object. We also split the code into specialized files for readability and organization.

Calculation Details: Line Smoothing

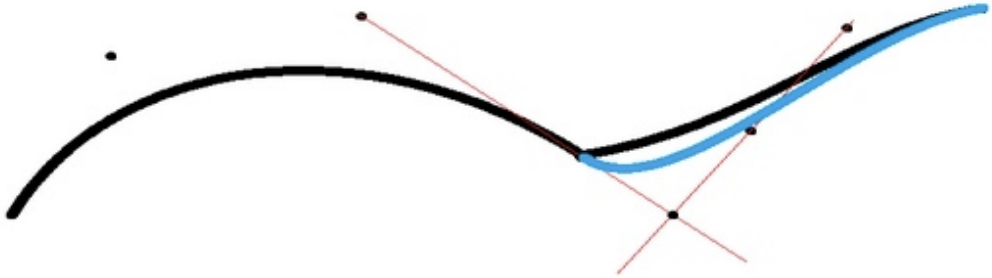
- Done with Bezier curves drawn from four control points.
- Control points are adjusted so that the first control segment defining a curve is colinear with the last segment of the previous curve.
- Done by adjusting the second control point of every curve to be equal to the intersection of the of the above two segments.
- Points Px and Py in the below equation are the intersections of two lines defined by two points each.

Calculation Details: Zoom

- Upon a click, scale the canvas and apply a translation so that the ratios of the mouse position and the edges of the visible canvas don’t change.

- This is done by first applying a scale to the canvas by some zoom value and then applying a transform by a value dependent upon where the mouse was clicked.
- This is done by first evaluating the true position on the canvas the mouse was clicked: x = (current translation)/(current zoom) + (click coordinate)
- The new translation is calculated by the equation: (new translation) = x/(new zoom) - x.

$$(P_x, P_y) = \left(\frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \right)$$



Calculation Details: Fill Tool

The fill tool was implemented in several different ways before arriving at an optimal solution.

- *Pixel Method*: In most implementations of a flood fill, coordinate data is found using a BFS algorithm using all pixels as connected nodes. This results in a set of pixels encompassing the fill. When implemented, however, saving this data proved to be expensive, and rendering each pixel one by one using RGBa data hurt performance.
- *Segment Method*: In this method, the fill area was broken down into segments one pixel high. When searching a segment, the algorithm “searches” East and West until it cannot expand any further. It then saves the endpoints of the segment and, when rendering, draws a line between them (faster than specifying data for each pixel). In addition, while searching, the algorithm looks North and South and determines how many segments lie above and below. For each segment, it generates a

"seed", which is used as the starting point for the next segment. The segment method improved upon space and performance, but suffered in a new way; when the fill object was rotated or scaled, spaces between the segments showed.

- *Sector Method:* The sector method expanded upon the segment method; in this method, the goal was to break down the area into "sectors", which were closed sections of the area that could be defined by coordinates. The segment method searched as usual, but saved the Eastern point in an array call "lPts", and the western point in an array called "rPts". If there was only one segment above or below this segment, then the sector continued to expand. If there were zero or more than one segments, then the sector was closed, and lPts was concatenated with rPts to create a complete set of exterior points. Any extra segments became seeds for new sectors. Although significantly more difficult to implement than the segment method, the sector method improved greatly on the quality of the fill. Unfortunately, overlap and separation of sectors was an issue in semi-transparent fills, which led to the next implementation.
- *One Big Huge Sector Method:* The one big huge sector method was, again, an expansion on the previous method. Given a set of sectors defining a fill area, it attempted to connect them together to create one large fill area. The algorithm designed to do this was recursive, and took advantage of Javascript's array structure. Each sector was assigned a unique id. The algorithm would begin traversing a sector's lPts array, adding each point to a "master list". If a sector was "connected" to another sector, it's partner's id would take the place of a point in its lPts array. When this id is encountered, the algorithm calls itself on the partner sector (reversing the order of the points if needed), before continuing. The one big huge sector method unfortunately proved no improvement over the sector method, and the recursive formula added overhead to the computation, so it was abandoned in favor of the sector method.

Anti-Aliasing:

- Regardless of the method that was used, the fill tool relied on the comparison of colors to determine whether or not a pixel was "fillable". If a user clicked on a white pixel, the fill would expand to cover all white pixels within that space. This is how all fill tools work. Unfortunately, the HTML5 canvas presented an issue; when objects were drawn on the canvas, an "anti-aliasing" effect was automatically applied. Whenever there was a border between two colored areas, the colors were blended

slightly to make the line appear less pixelated. For the fill tool, this became a major issue, as the anti-aliased region was detected as not “fillable”, and was discarded. All fills, therefore, were not complete, as there was an anti-aliased region around all of them. As of right now, Javascript and HTML5 offer no possible way to turn off anti-aliasing. This lead to a variety of attempts to overcome the anti-aliasing, including calculating “change rates” between each pixel and its neighbors, in an attempt to detect anti-aliased regions. Unfortunately, the anti-alias solution was still incomplete at the end of the project.