BIMSHIFT

HTML5 Data Visualization and Manipulation Tool

Colorado School of Mines
Field Session Summer 2013

Riley Moses
Bri Fidder
Jon Lewis

Introduction & Product Vision

BIMShift is a company that provides all-encompassing building information modeling services including design, construction, and management. DataGate is their user-facing portal that allows for pushing and pulling building information from AutoDesk Revit. Clients can keep track of everything about a building: individual floor plans, machine and part information, pipe locations, and much more.

Users can view 2D and 3D representations of floor plans along with their associated information, such as which rooms belong to which departments. If a pipe breaks, clients can look up the specific part online without needing to physically inspect it. Prior to this summer, all information could only be viewed via static images in PDF documents. To edit information, users would have to navigate to a separate part of the site.

The primary goal of this summer field session was to provide BIMShift with a unified interactive tool for viewing and editing information about these floor plans from within a web browser. Rather than viewing a static image, our tool allows the user to dynamically add and update the information associated with floor plans by drawing polygons and associating information with them. It integrates with BIMShift's existing software infrastructure and provides a consistent experience on both desktop and mobile browsers.

Requirements

I. Functional

- Extract image from a PDF and display it in a web page
- Draw freeform polygons on the image
- Associate information with the polygons via pop up
- Save information to BIMShift’s existing database
- Load existing information from the database and display it
- Edit existing polygons and associated information
- Keep track of changes made by the user and confirm them before saving

II. Non-Functional Requirements

- Integrate with all of BIMShift’s existing PHP and JavaScript code
- The tool must be usable in Firefox, Chrome, and IE on the desktop
- On iOS, the tool must be usable in Safari. On Android, the tool must be usable in at least Chrome or Firefox
- To handle interaction with large and complex images, the user should be able to zoom in and out
- The status of shapes (whether or not they have been saved) must be easily understood
- It should be clear which shape has been selected when editing information
- Ensure the database is always in a consistent state
- Minimize the number of database calls
- Do not allow any interaction while saving or loading

System Architecture

Our tool is essentially a self-contained web application integrated with the rest of BIMShift's code. All of the user-facing elements are handled via HTML and JavaScript, including state management and user interaction. The user draws points to create polygons, those polygons are associated with real-world elements and then saved as part of an overlay for a specific sheet (the selected floor plan). Data is then sent and retrieved to/from the database using AJAX calls to PHP scripts. See Figure 1 below.

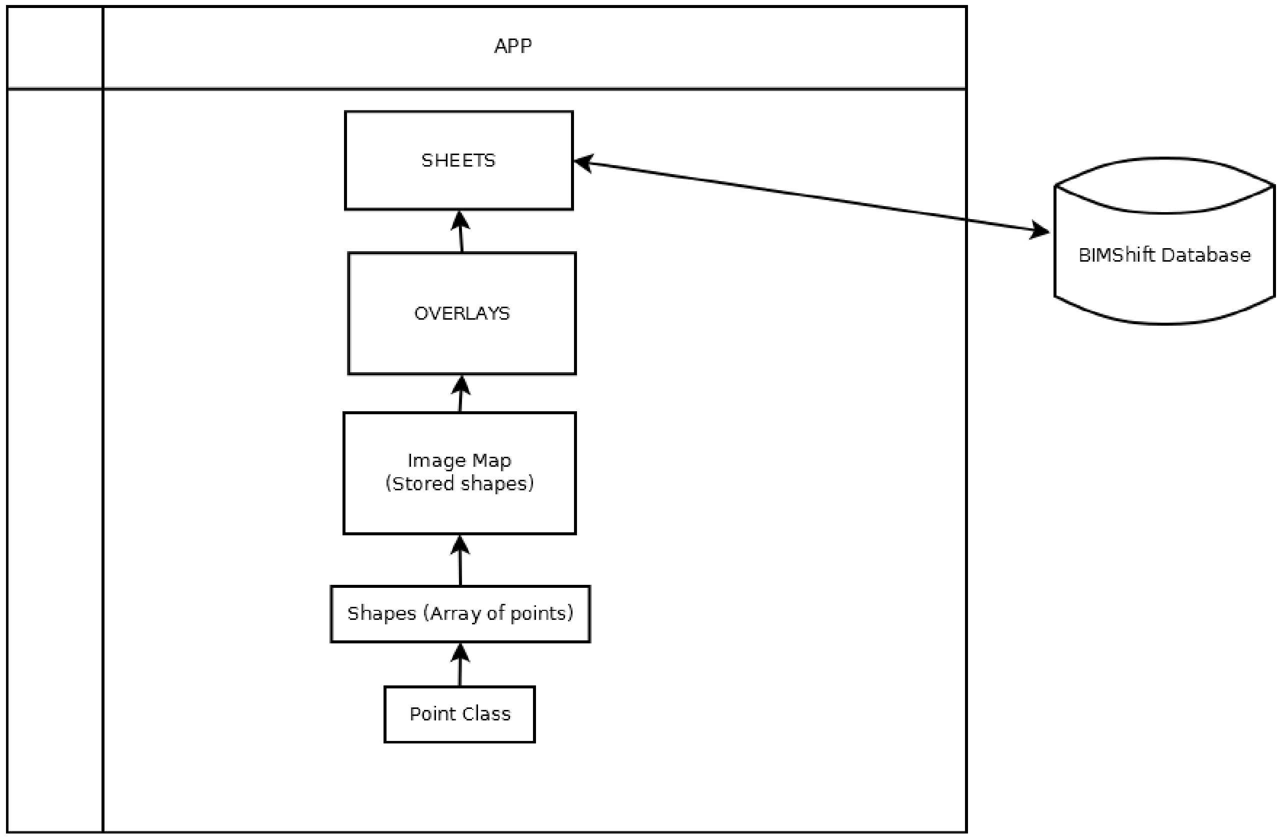

Figure 1: General Architecture

Technical Design

Everything in KineticJS is done using layers, which collectively make up the “stage.” Layers can contain individual shapes, groups of shapes, and groups of groups. Layers, groups, shapes, and the stage itself are all treated as separate “nodes,” and can be styled and transformed independently. Our project is set up with three layers, as shown in Figure 2: One for preparing while the page is loading, one for the image being viewed, and one for the shapes being drawn.

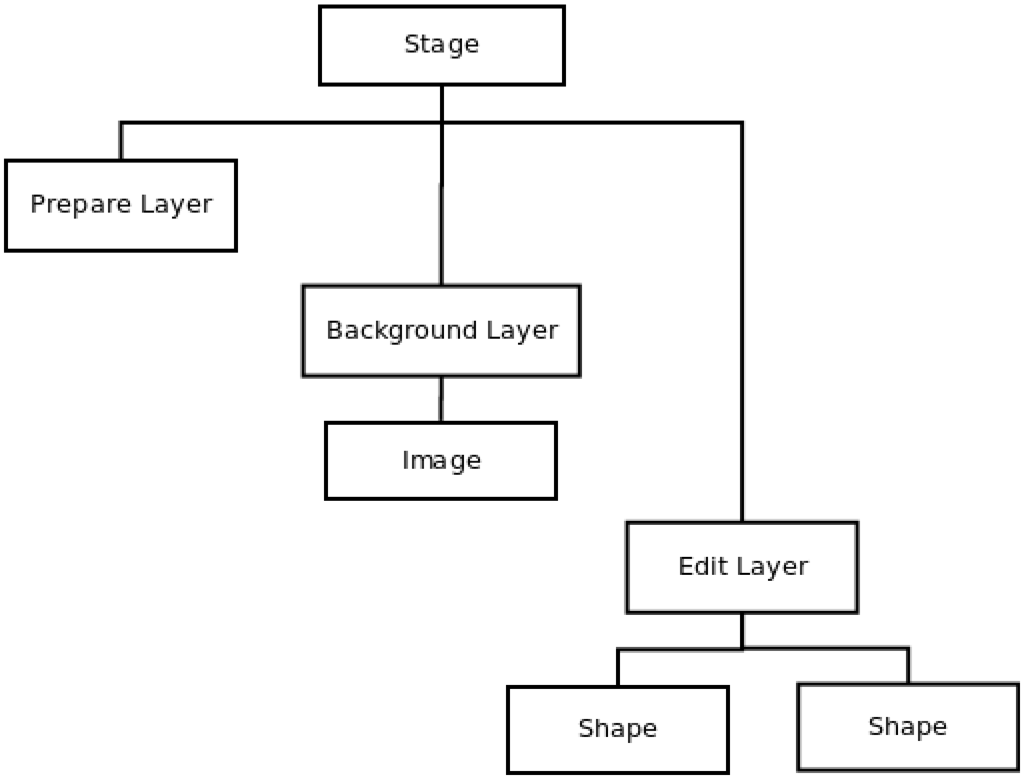

Figure 2: Kinetic layers in our project

The prepare layer is used while the page is loading, and prevents any interaction with the canvas while displaying a “Preparing assets...” message. Once the image has been loaded, it is added to the background layer. The prepare layer is then replaced by the background layer. After querying the database, any polygons that already exist are added to the edit layer and it is displayed on top of the image. Any shapes drawn by the user are added to the edit layer. All of the layers are transformed when the user translates or zooms, so the locations of shapes is always consistent with the image beneath them. Events are handled by the stage or by individual shapes.

The tools available to the user to edit the shapes being drawn varied widely based on what mode they were in. So, our solution was to instead sort functionality in the program based on what tool the user was using, and focusing on making the entire program a state machine.

After initial load, the program cycles through four different functions every time they change tools, which accomplish the designated task based on what state the user is in. These four functions are prep, draw, save, and cleanup.

When the user selects a new tool, the first function to run is cleanup, which removes any of the state variables that were used by the old state, as well as removes any unfinished elements that the user may have started but not completed. After cleanup, the program then enters the state prep function, which prepares any of the functions or hooks needed by that tool, as well as setting up initial variables for that tool's usage, such as assigning the various listeners to shapes when entering edit mode. Any time a user makes a change and saves that change to the system, the program then runs its save function, which is very similar to cleanup, but commits their change to the list of shapes to be saved, instead of removing them from view. Then, when the user selects another tool, the process repeats itself. All over again.

One of the main advantages to this design is that it allowed us to very specifically delegate behavior to tools based on what mode they were in, instead of having massive numbers of logic flows that would have been both difficult to debug and prone to error. This also allowed us to arrange the code based on what we wanted to accomplish, which was much more intuitive than sorting the code based on when the task would have occurred.

The disadvantage to the code being structured this way is that is was very difficult to insure that the code remained DRY, as it was very easy to write similar behavior for multiple tools without realizing that the code existed elsewhere in the program, as well. One way we overcame this design problem is with heavy usage of utility functions, and delegating responsibility of behavior instead to these. One example of this is a function we created that was responsible for toggling the buttons to the correct state. The buttons were outside the scope of the tool's functionality, but it was still necessary for us to have buttons enabled or disabled depending on the tool that the user had active.

Initially, we left the prep, draw, save, and cleanup buttons responsible for button activation, but this was leaving us with buttons that were frequently disabled or enabled inappropriately, or extra buttons were added that really didn't need to be. Instead, we have the program call the function that toggles buttons to set properly any time one of them needs to be changed, and then trust that the function will do its job as intended. One major advantage to doing this was that, for the most part, categories and patterns of when buttons were enabled or disabled became apparent very quickly, and a one-line solution could be implemented to accomplish the task.

There are four main tables in the database for our project, The overlays table, the polygons table, the overlay associations table, and the coordinates table. The overlays table connects each sheet to the user editing it and each overlay has a name attribute as well. The polygons table stores the shapes with additional information, such as the file id, the element id, the graphic (if it exists), comments the user can add to the shape, and etcinfo which is additional information about the polygon (for example, if it is a circle or not). The coordinates table is where all of the x and y positions as well as the sequence of the points of each polygon are stored. The sequence of the points was necessary to store because that's how the points were sorted when loaded from the database to create the correct shape. Each coordinate has a foreign key of a polygon id that it belongs to. Additionally, there is a join table for the polygons and overlays so that multiple polygons can belong to multiple overlays and vise versa. The entire schema can be seen in Figure 3.

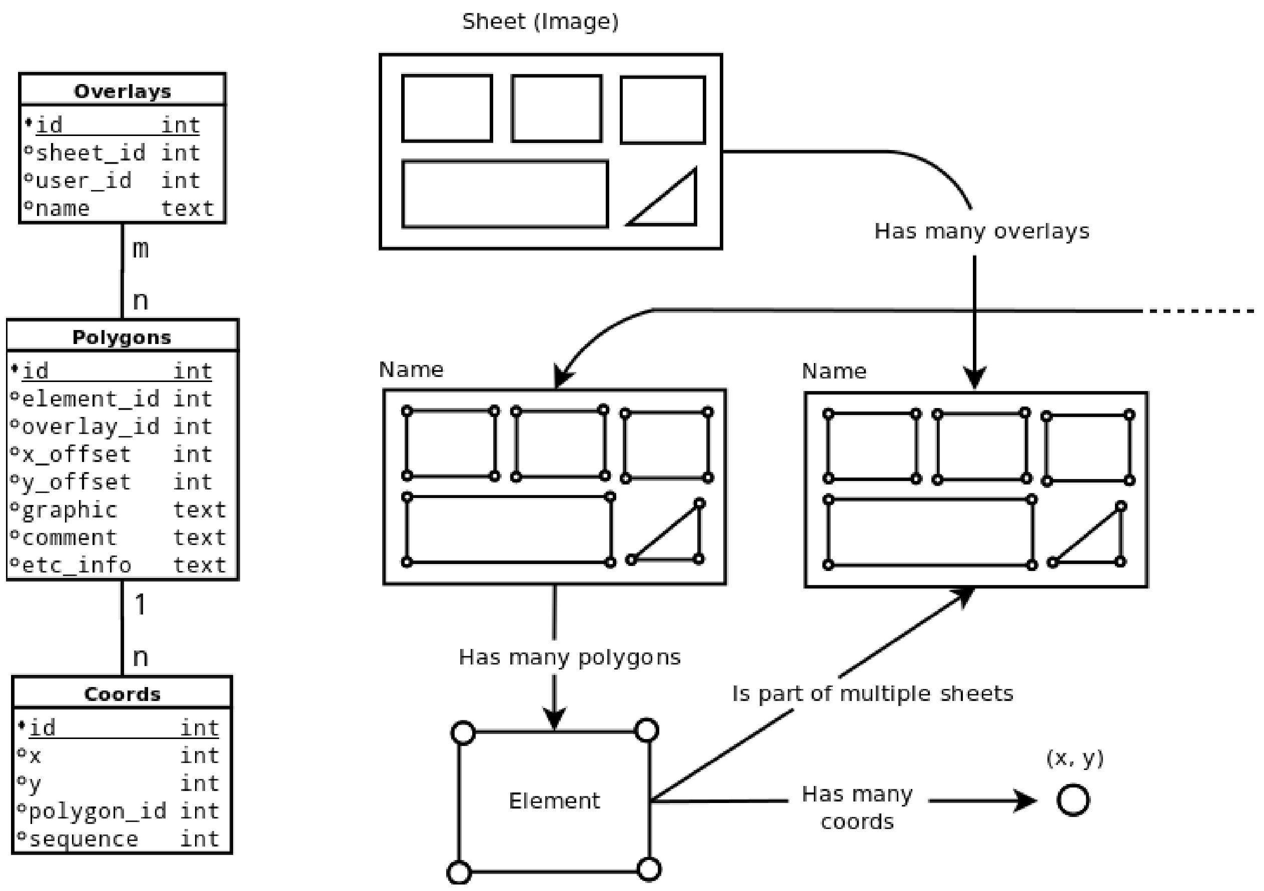

Figure 3: Database schema

Design & Implementation Decisions

We opted to use the KineticJS JavaScript library for all of our canvas-related code because it allows us to easily draw and manipulate what is shown to the user. It provides a “stage” on which everything is drawn, and this stage is what we scale when the user zooms in or out. All of the graphics are added to layers which are displayed on top of the stage and scale along with it. This minimizes the amount of work that we have to do directly with the HTML canvases, instead allowing us to think of everything in terms of the stage and the layers on top of it. KineticJS also includes classes for polygons, so almost every single graphical element on the canvas maps to an associated object in the code. Rather than figuring out ourselves if the coordinates of a user’s click fell within the bounds of a shape, KineticJS allows us to attach listeners to each individual entity. The user’s actions can be directly mapped to methods that manipulate the shape objects.

We use ImageMagick to extract images from PDF, called using the PHP `exec` function via AJAX rather than using plugin. We decided to call ImageMagick directly because it gave us a higher resolution image than the plugin did, and we often encountered inconsistent behavior with ImageMagick PHP plugins. As an added bonus, calling ImageMagick directly resulted in faster execution, because we were able to call the 64-bit version of the utility.

We decided to disabled touch zoom on the Chrome browser on Android. We use double-clicking to zoom in on a point by a set amount. When the user places two fingers on the screen at the same time, regardless of the time between the touches, Chrome triggers a double-click event. However, because we also allow pinch-zooming on touchscreen devices, firing this event causes the stage to zoom in before the user ever moves their fingers. No other browser treats a two-finger touch as a double-click, so we decided to selectively disable double-tap zooming in Chrome on Android.

We treat anything drawn by the user as a polygon, regardless of number of points. We made the decision to design the code this way so that it would reuse code for different shapes and make the code more dry and less repetitive. The client wants us to add in single point functionality and having all of the shapes being treated the same makes adding that easy.

We chose to structure the program as a state machine, where the user always exists in a certain state, or ‘tool’ they have selected, and then program functionality is dependent on what tool the user has chosen. Figure 4 shows the how polygon mode causes anchors to appear. The behavior of the drawing tool is determined by the state that the user is in. Depending on what tool the user has selected, the code then flows through three primary methods for all tools: cleanup, prep, and draw functions. The cleanup step is responsible for removing any elements that are used by that state specifically, and returning the state variables and global settings to a “clear and default” setting. After this, the program enters the prep function, where any global parameters for the new state are set as needed and global settings are adjusted.

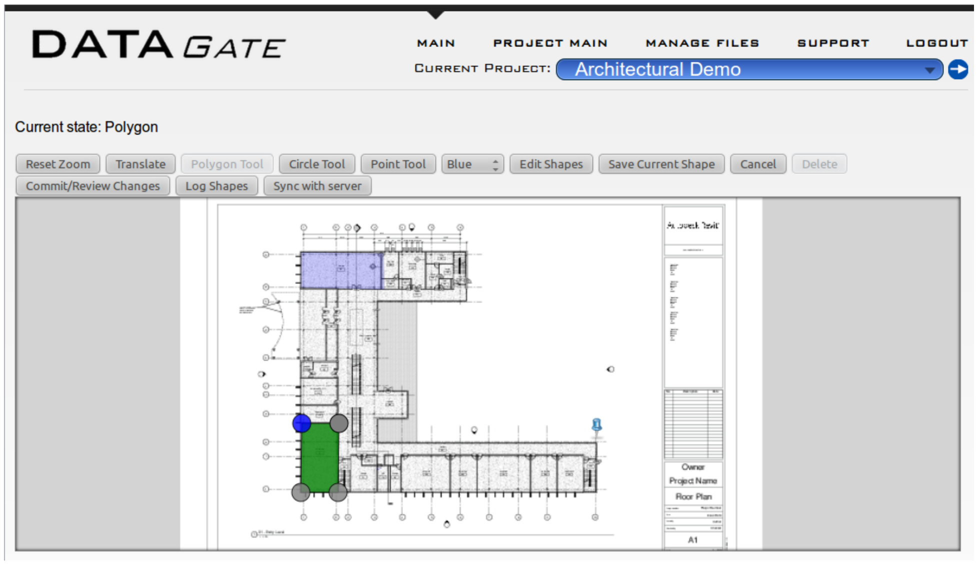

Figure 4: Drawing a polygon. The blue anchor is the starting point.
A saved polygon and a blue pin can also be seen.

This program structure was chosen because of the way JavaScript uses listeners. Pushing a button or clicking triggers a function call, but one interactive object can only call one function. Since only our main canvas really has one global listener, all behavior for that object is entirely determined by what tool state the user has selected.

One exception to the main canvas using a global listener is polygons are responsible for their own behavior in edit mode, and have internal listeners. This choice was made because our framework, KineticJS, has a very strong embedded method for objects listening for their own internal clicks, but determining externally if the user is clicking on a polygon was a much more difficult task in comparison

Results

The goal of this project was to create a drawing program that would interact with floor plans selected by the user to create regions with element data tied to them. It has been tested in all major browsers including mobile ones on both Android and Apple.The only browsers that it will not function in are older browsers that do not support HTML5.

Our project meets almost all of the functional requirements specified by the client. The only missing feature is the ability to customize the fields shown in the pop-up, but that functionality was determined to be outside the scope of this project because the pop-up will display the contents of a separate web page. We successfully authorize the user, extract the image from the PDF, allow the user to draw polygons on the image using HTML5, allow the user to modify polygons and associate information with them, record changes, and save everything in the database.

Extra features like dynamically updating the floor plan image and editing the colors of polygons did not make it into the final release due to time constraints. There are hooks in the code for supporting multiple overlays, but that was also extra functionality and was not implemented in the interface.

Besides building a working product, we also had several lessons that we've learned that should help us with workflow on future projects. Before using this project, some members of our group would directly modify master when adding a new feature. Before long, some of the features we were adding would cause some instability in the product, and this made it difficult to test them. After this, we came to the conclusion that we should keep the master branch in a working state, and should merge features after they have been tested and are stable.

Another lesson learned is that although we had the capability of working from home, all of us found that we were significantly more productive when at the workplace, despite the commute. Shared workspaces have also proven to be immensely helpful, and after having a short while where we were looking at each others' monitors, we found sitting at the same desk to be impractical, and set up a remote desktop sharing system to that we could see the same screen.

One further way this system could be improved is if the shared monitor was on a third computer, and then both users would have equal access to it, instead of someone sacrificing usage of their computer to have it. It was also nice to learn about frameworks for working with the HTML5 canvas. In our case, KineticJS made drawing, manipulating, and saving all of the graphical information much easier than it would have been if we had directly manipulated a single canvas. JavaScript has also proven to be very hard to debug.

Appendices

Further Information about the libraries used in this project can be found at the links below:

- KineticJS:
  http://kineticjs.com/
  https://github.com/ericdrowell/KineticJS
- KineticJS and HTML5 Canvas:
  http://www.html5canvastutorials.com/kineticjs/html5-canvas-events-tutorials-introduction-with-kineticjs/
- Imagemagick:
  http://www.imagemagick.org/script/index.php