Datagraphics creation for the modern web

A proof of concept

Tom van de Zande Louis Smit

December 21, 2012

1. Preface

This is the bachelor's thesis of Tom van de Zande and Louis Smit for the bachelor of science program in Computer Science at the Delft University of Technology. In the months September through December of 2012 we have examined the creation of datagraphics for the modern web environment. A task commissioned by Attingo Services B.V. Attingo has an advisory role in Schwandt Infographics, a company specialized in creating infographics. In this role Attingo facilitated the project.

We would like to thank our supervisor, assistant professor Dr. Martin Pinzger from the Software Engineering Research Group at Delft University of Technology. He stimulated and supervised us in our ambition to use modern software development methodologies whilst giving us the space to experiment and discover things on our own.

A special thanks also goes to our mentor Wichert Akkerman, CTO of Attingo, who closely monitored our progress and helped to set realistic yet ambitious goals.

Contents

1. Preface **2**

2. Introduction **5**

3. Problem analysis **6**
3.1. Assignment . . . 6
3.1.1. Datagraphics . . . 6
3.2. State-of-the-art . . . 6
3.3. User stories . . . 7
3.3.1. SVG Graphics . . . 7
3.3.2. Data binding . . . 7
3.3.3. Interaction . . . 8
3.3.4. Transitions . . . 8
3.3.5. Platform . . . 9

4. Concepts and solutions **10**
4.1. Scalable Vector Graphics . . . 10
4.2. Transitions . . . 11
4.2.1. Modifiers . . . 12
4.3. Interaction . . . 12
4.3.1. UI-control states . . . 12
4.3.2. Actions . . . 13
4.4. Data-binding . . . 13
4.5. Data models . . . 14

5. Architecture **15**
5.1. Information viewpoint . . . 15
5.1.1. Projects . . . 15
5.1.2. Scenes . . . 15
5.1.3. Elements . . . 16
5.1.4. DataModels . . . 16
5.2. Functional viewpoint . . . 16
5.2.1. Parser . . . 16
5.2.2. Backend and ViewerState . . . 17
5.2.3. DataPoller . . . 18
5.2.4. Viewer . . . 18
5.2.5. Editor . . . 18
5.3. Concurrency viewpoint . . . 19
5.3.1. Pitfalls . . . 20
5.3.2. Future considerations . . . 20
5.4. Development viewpoint . . . 20
5.4.1. Development environment . . . 21
5.4.2. Testing and developer workflow . . . 22

6. Recommendations **24**

7. Process **26**
7.1. Scrum 26
7.2. Reflection 26

8. Conclusion **27**

A. Original assignment (in Dutch) **29**
A.1. Project voorstel 29
A.1.1. Opdrachtgever 29
A.1.2. Opdracht 29
A.1.3. Use case 30

2. Introduction

This document contains the findings of a feasibility study of an application enabling infographic [1] designers to convert vector drawings into interactive, data-driven web applications, a.k.a. datagraphics. In our view both content and form are equally important in datagraphics, which is why infographic designers are preeminently suited to create them. Traditionally, the Adobe Flash platform was a popular choice since it did not require extensive technical knowledge or programming skills. Although quite suited for animations and interaction, implementing data-driven graphics in Flash is still very hard to do. In general, it is not very adequate for content. In recent years there has been a clear shift towards new technologies, which mostly can be shared under the umbrella of HTML5 [14]. These technologies have great potential. They offer flexibility, performance, compatibility and have the support of industry leaders like Google and Apple. Unfortunately though, these new techniques are even less accessible for non-programmers. Our goal is to create an application that will bridge the gap between infographic designers and state-of-the-art technologies, specifically in service of creating datagraphics.

The research was carried out by means of developing a proof of concept. This document describes our analyses, solutions and results of this process. First we will analyze and describe the problem in Chapter 3. Then, in Chapter 4 we introduce concepts and solutions at the basis of our application. Chapter 5 describes our software architecture from different viewpoints. This chapter is especially of value for future developers who want to continue the work. Next we describe our methods and reflect briefly on the process as a whole in Chapter 7. In Chapter 6 we put forth our recommendations on how to proceed. Finally, in Chapter 8 we present the conclusions of this project.

[1] Information graphics or infographics are graphic visual representations of information, data or knowledge intended to present complex information quickly and clearly. [9]

3. Problem analysis

We are going to describe the problem space in terms of user stories. These are used to get a better sense of the desired capabilities of the product. But first we will clarify the assignment and look at competing products.

3.1. Assignment

The original assignment, as defined in collaboration with the client, can be found in the Appendix A.1. In short it describes a prototype enabling infographic designers to turn their vector drawings into interactive, data-driven web applications or datagraphics. In this section we will elaborate further on what this means and how we intend to provide for this need.

3.1.1. Datagraphics

In order to get a better feeling for the term datagraphic, it might help to understand better the circumstances in which the demand has arisen. Schwandt Infographics (target of the product to be developed) states on its website: "We specialize in providing insight into complex information and abstract processes. **Form and content** are thereby **equally important**." Until recent years they realized this mainly in the form of static graphics. However, new opportunities arise when designing for the internet. Graphics no longer need to be static:

- Information can be displayed in different views and layers;
- Users can interact with it;
- Simple animations can be added both in service of form and content;
- Aspects of the graphic change in real-time as data changes. A bar in a bar chart would be the simplest example. But there are multiple ways that visual representation of data in the graphic could be bound to a data point. This concept is called **data binding**.

Datagraphics should provide in all these things (and thus not only, nor particularly the last point). We mention this because we want our application to be in service of both content AND form equally, since there is no reason to assume Schwandt Infographics would suddenly change its values.

3.2. State-of-the-art

To get a better understanding of why Schwandt Infographics' needs are not being met by current offerings, we will look at what similar/competing products exist in the market today.

On april 12, 2012, Adobe discontinued a product called Flash Catalyst, which is the closest we've seen to what we're proposing. It allowed a designer to import graphics from Adobe Illustrator and add interactivity and transitions to elements. However, for data binding, the user needs to switch to another Adobe product called Flash builder, which required the user to write code in ActionScript and Adobe Flex. Aside from it only outputting in the Flash format, this is a work flow we find unacceptable for straightforward data graphics.

More recently Adobe also seems to embrace HTML5/JavaScript in favor of Flash and released a set of tools called Adobe Edge. Since this is a new environment for Adobe these tools are still pretty limited, but one product that jumps out is Edge Animate. It is an animation tool for HTML elements, although it does not support SVG. The animations itself are created using CSS and jQuery and programming skills are still required. An interesting choice by Adobe, since much richer animations can be achieved by the SVG and Canvas web standards.

There is also Adobe Muse, a tool geared towards designer's that want to create a website. Although closer to our intended audience, the limited transitions and lack of data binding render this product inadequate.

Some startups are also venturing into the space. Easel.io provides an editor to arrange standard HTML elements and create mockups / web applications. Visual.ly has the same goals as this project, but has so far only released a social network for static infographics. Their progress remains unknown.

To us, it is clear the decline of Flash has left a void in the way designers can build interactive content. Moreover, no other product on the market seems to focus on providing a user friendly way to create data bindings.

3.3. User stories

In this chapter we'll look at the different aspects of the product. Each section will list some relevant user stories after which we will discuss the possibilities and difficulties associated with those stories. Not all stories have come to fruition during the course of this project due to its limited timespan. Bolded stories are part of the intended minimum viable product (MVP). The rest are non-essential, or, nice-to-haves.

3.3.1. SVG Graphics

An important aspect of this project is the way in which vector graphics make their way from the designers pen (tool) to a point where they can be easily manipulated to add interactions, transitions and data bindings. An important focus for us was to strongly keep in mind the current workflow for static infographics and try to be as least invasive as possible.

User stories

As a designer of the datagraphic

1. ***I want to keep using the vector editing product I've built my skills for (Adobe Illustrator).***

Adobe Illustrator is the leading vector editing product that has been crafted for more than a decade by a large team. It seems ambitious to say the least to replicate even a small part of what Illustrator can do. What this means is that this prototype is not a vector graphics **editor** but a vector graphics **interpreter**. This has important implications. We need to parse vector graphics so we can identify individual elements, which can than be manipulated later on. What this also implies is that we are giving away control over a big part of the datagraphic creation process to another software product. On the plus side, this means we can streamline the prototype to be just an editor for transitions, interactions and data binding.

3.3.2. Data binding

The difficulties associated with data binding are hinged upon the complexity of the data. We will need to constrain the types of data the prototype can handle in order to finish the project in a timely manner.

User stories

As a designer of the datagraphic

1. ***I want to bind certain attributes of a graphical element, like size, color and position, to an external data point.***
2. ***I want to manage the (dummy) data myself, so I can play around with different values.***
3. *I want to be able to work with a programmer on more complex data visualizations.*

As a software developer working on a datagraphic

1. *I want a well documented API to develop against, so I can work together with the designer on more complex data visualizations.*

The high level concept here is that we have a data model that should have a visual representation in the graphic. The domain of the data model needs to be mapped to a range of states of the graphical element. This can be both in a discrete and a continuous space. A continuous example would again be a bar in a bar chart that adjusts its height to a number. A discrete case would be where you want to visualize the state of a real world object which possible states are known. For example, a data center could have an overview of all its servers in a datagraphic and the representation of each server could change based on the status of the server (running, shut down, crashed etc.).

The designers third story and the developers first, reference the widget API as originally described in the assignment. Providing a safe API requires quite some effort and interferes with streamlining the proof of concept and keeping it as simple as possible, while still producing relevant insights. However, a widget API would be a tremendous addition to the usefulness of the product and should definitely be researched.

3.3.3. Interaction

In our definition, datagraphics can be interactive.

User stories

As a designer of the datagraphic

1. ***I want to allow the user to navigate to different views of the datagraphic.***
2. ***I want to allow the user to activate sub views like 'tooltips' and additional information panels.***
3. ***I want to allow the user to select different views on the same data.***
4. ***I want to allow the user to select different data for the same view.***

There are varying degrees of complexity here. The difficulty here is not so much the UI-controls themselves, but letting the designer define what happens when they are activated.

3.3.4. Transitions

An integral aspect of these interactive datagraphics is some sort of animated transition.

User stories

As an end-user of the datagraphic

1. ***I want transitions to take place when I interact with the datagraphic.***
2. ***I want transitions to take place when data and its corresponding view changes.***

As a designer of the datagraphic

1. *I want to manage the timing properties of all transitions, such as delay, duration and easing function.*
2. *I want to be able to define a custom transition for when en element gets added to or removed from a scene. For example: fade in/out, slide in/out etc..*
3. *I want to manage the transition when an element changes its form from one scene to the next.*
4. *I want to manage the transition for any element that changes state, like a button.*
5. *I want to manage positional transitions, such that the element moves along a path I can define.*

Doing all these things will mean a significant time investment, so we'll need to need a subset that adequately proves these things are possible. For the MVP, having a default transition goes a long way, but it would be preferable to be able to configure at least the duration and delay of transitions. We also need at least one custom transition for when elements are added to or removed from the scene.

3.3.5. Platform

Since the employer left the choice of platform to us, we will discuss the environment of the application, be it on the web, desktop or perhaps mobile.

User stories

As an end-user of the datagraphic

1. ***I want to explore the datagraphic in a browser.***
2. ***I want to explore the datagraphic on a mobile device.***

As a designer of the datagraphic

1. *I want to edit the datagraphic offline.*
2. ***I want to be able to collaborate with other designers in real-time.***

As the developer maintaining the software

1. ***I want to be able to easily push bug fixes and upgrades to the server.***

Since the end-product (the datagraphic) will run in a web browser, it seems a logical choice to have the editor run there as well. The only interesting story here is the ability to work offline. As mentioned earlier, static graphic files will need to be parsed and the elements in it persisted. Because the browser is sand-boxed, this is a big challenge, but not impossible. Having an online requirement will be much easier for the prototype however, as we can parse and persist on the server. As a bonus, all application state can be relatively easily synced to all clients working on the same project.

4. Concepts and solutions

This chapter introduces core concepts and solutions that provide the basis for interactive and data-driven visualizations in the web environment.

4.1. Scalable Vector Graphics

One of the main reasons we originally thought creating datagraphics in the browser might be feasible was the fact that HTML5 specifies native support for Scalable Vector Graphics [14]. The same format [15] most vector applications like Adobe Illustrator can export to. This also influenced the decision not to develop any vector editing capabilities. Instead SVG files are created externally and uploaded to the server. In order to understand the process of converting SVG files to datagraphics it is essential to have some insight into the SVG file format.

SVG is an XML-based file format. Listing 4.1 shows the simplest possible SVG file containing just a rectangle. Figure 4.1 shows the same graphic when rendered.

Listing 4.1: A very simple SVG file

```
<?xml version="1.0" encoding="UTF-8" ?>
<svg xmlns="http://www.w3.org/2000/svg">
    <rect width="75" height="75" />
</svg>
```

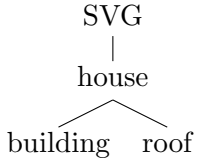
Figure 4.1.: Rendered graphics from Listing 4.1

As a result of being XML-based, SVG has some principled properties. The most significant being:

- Elements are defined only by their type and attributes;
- Documents have a tree-like structure.

The first is used to determine the differences between two elements, which is crucial for doing transitions as shall be explained in Section 4.2. The second is essential for the way we render elements as described in Section 5.2.4.

Listing 4.2 gives a better impression of these properties. It is a drawing of a house made up of two separate shapes or sub elements; the building and the roof. The tree structure therefore could be displayed like this:



Listing 4.2 also shows that both the roof and the building are *path* elements only different in their attributes. In this case just a *d* attribute containing the path data. This is a very typical example, because it turns out most drawings are to a large extend, made up of paths. As a result it is very hard to understand the graphic without rendering it. For this reason we ask the designer to define id attributes for elements they want to give certain behavior in our application. Similar to the way the building and the roof are identified in the house example.

Listing 4.2: A simple drawing of a house

```
<?xml version="1.0" encoding="UTF-8" ?>
<svg xmlns="http://www.w3.org/2000/svg" >
  <g id="house" >

    <path id="building" d="m 27.224831,72.958694 c 31.531036,0 63.062072,0 94.593109,0
         0,24.962072 0,49.924146 0,74.886216 -31.531037,0 -63.062073,0 -94.593109,0
        0,-24.96207 0,-49.924144 0,-74.886216 z" />

    <path id="roof" d="M 2.2040278,73.509396 C 25.915674,49.797749 49.627321,26.086103
         73.338967,2.3744558 97.050618,26.086103 120.76227,49.797749
        144.47392,73.509396" />

  </g>
</svg>
```

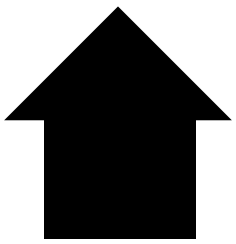


Figure 4.2.: Rendered graphics from Listing 4.2

4.2. Transitions

Transitions are a core concept of a datagraphic. They constitute one of the essential differences between infographics and datagraphics. An infographic can be defined as a static graphic providing information, while a datagraphic is dynamic and able to show many views on potentially changing information. To change from one view to another, some kind of transition is required.

In our application, a predefined view of the datagraphic is called a scene. Scenes can be seen as the equivalent of a slide of a presentation. When navigating from one scene to the next a transition takes place. Some elements might appear and others disappear. More interestingly, some elements might change. Figure 4.3 shows two scenes containing the same element; a red circle, c1. When navigating from scene1 to scene2, the differences between these scenes are used to provide a smooth transition. In this case the only difference is the position of c1 and as a result it would animate from the top left corner to the bottom right.

Scenes are in essence top level states of the datagraphic. Certain elements **within** a scene can be stateful too. For example a button has a standard state and a hover state (with a different appearance). The hover state is visible when the mouse cursor is hovered over the button. More on this in Section 4.3. As will become clear shortly, the same mechanisms apply to the changing of scenes as to the changing of states of stateful elements. Every element has a *context*. This

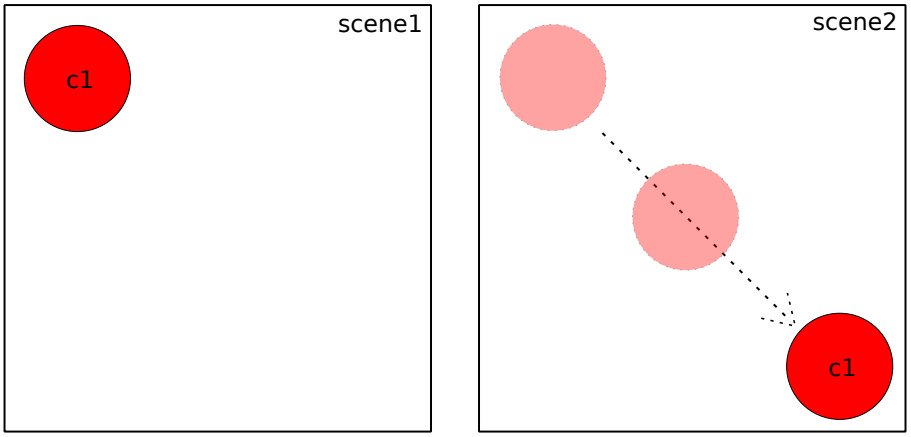


Figure 4.3.: When navigating from scene1 to scene2, c1 transitions from the top left to the bottom right corner.

can be either a scene or a state of a container element. When an element is stateful, its children have this element as their context.

This means that the same element can have a different appearance in a different context. Lets call these different appearances, **instances**. This is where transitions come in, to provide for the change from one context to another. And thus to transform the instances in the first context to the instances in the second.

4.2.1. Modifiers

To have a transition between 2 instances take place, the differences should be known. A modifier is basically a function that can eliminate one difference between two instances. All modifiers together can perform the complete transformation from one instance to another. Since instances differ per context, modifiers are associated with the context of the new instance. This being the context in which those modifiers are active.

In Section 4.1 we explained SVG elements are defined only by their type and their attributes. Different instances of elements always have the same type but can have different attributes. As a result the persisted data of a modifier simply is an attribute name and its value. When the modifier is applied an interpolator is created that can calculate any value for the given attribute between the old and the new one. This interpolator is used to animate from one value to the next.

It is possible to apply more than one modifier at a time on an element. This way, a smooth, animated transition from one instance to another is made possible. [1]

4.3. Interaction

In order to enable the designer to provide interaction for users, user interface controls are required. The most obvious example of an UI-control is a simple button.

4.3.1. UI-control states

Most UI-controls have different states, in the case of a button the *standard* state and the *hover* state. The standard state is visible when no interaction is taking place and the hover state is visible when the user is moving the mouse cursor over the button. Since this is just a transition

[1] For the interpolation and animation the D3.js library is used [2].

as with the scene example of Figure 4.3, the same mechanism applies. This is illustrated in Figure 4.4. In this example the rectangle r1 is enlarged when "hovering".

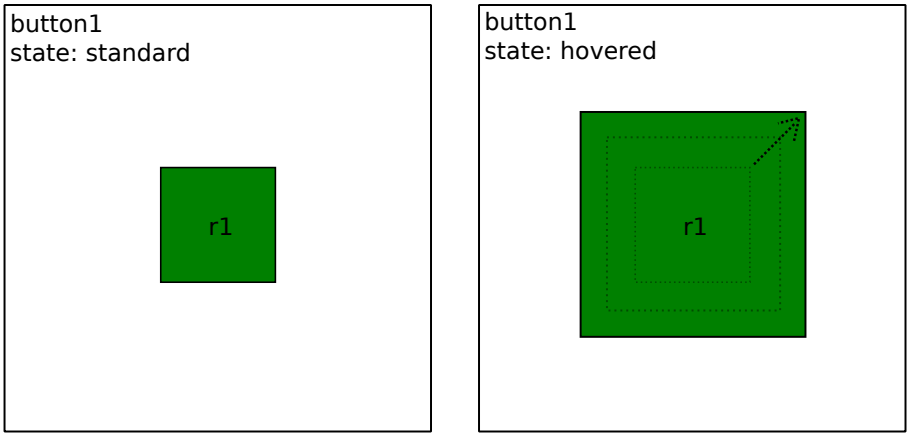


Figure 4.4.: Button state transitions

4.3.2. Actions

Most UI-controls set a certain process in motion when a particular interaction has occurred. Therefore the designer can add actions to user events. Events that can occur are dependent on the exact type of UI-control, but in the case of a button these would be *hover* and *click*.

The actions currently supported are:

- navigate to a scene;
- change state (of a stateful element);
- change data model (of a data bound element).

4.4. Data-binding

Data binding is needed when the appearance of an element depends on data. If an element is bound to a certain data model it is adjusted when the value of the model changes. To realize this, the domain of the data model needs to be mapped to a range of states of the graphical element. This can be both in a discrete and a continuous space. This way we can simply transition from state to state when the data changes. In the discrete case there is simply a one-to-one mapping from (data) values to (graphical) states and transitions happen in the same way we saw in the scene and button examples (Figure 4.3 and Figure 4.4). In this section we want to illustrate how a transition would work with a continues mapping.

In this example we have an arrow which length depends on a data model. The data model has a numerical value between a minimum and maximum. To accommodate the mapping from data to states, the designer defined two states for the arrow element; min and max. Obviously the minimum value of the data model can be mapped to the min state of the graphic and the maximum value to the max state. Our application however can use interpolation both of the data and the graphics to calculate any mapping in between. This process is illustrated in Figure 4.5.

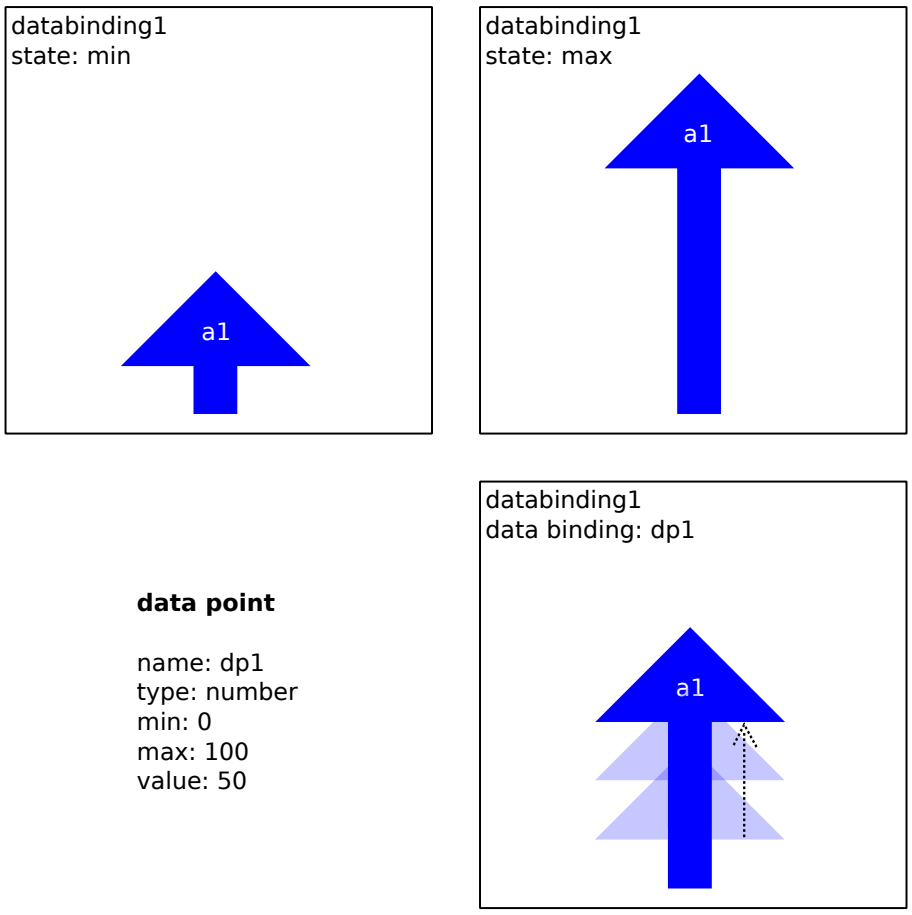


Figure 4.5.: Data binding to a number. A continues mapping between the data and the graphics is interpolated.

4.5. Data models

Obviously, in order to bind data to elements, there needs to be data first. For this proof of concept we decided to implement three basic data types:

- number;
- enumeration;
- plain text.

A number always lies between a predefined minimum and maximum value. An enumeration is one of a discrete amount predefined values. This could be either numbers, strings or a combination of those. Plain text is just that. These are three data types that were simple enough to implement in the limited time available and still are very useful. It becomes more complicated with data sets, where the visual representation is not a one-to-one mapping. There is a drop-off point where the data (binding) becomes too complex for a graphic designer to manage. Therefore, the goal is to provide simple data binding support and pawn off more complex ones to a programmer via the widget API.

5. Architecture

This chapter describes the software architecture from an information, functional, concurrency and development viewpoint. It is especially of value for future developers who want to continue the work.

5.1. Information viewpoint

Information management en flows were already discussed in the Functional viewpoint. However, the actual data models are not touched upon yet. Without going too much into implementation details, we do feel that providing some insights into the main data models improves general understanding of the application.

As shown in Figure 5.1 *Projects* consists of *Scenes* which in turn consists of *Elements*. An Element can potentially have a *DataModel*. Furthermore there are *Stateful* elements and *UIControls*.

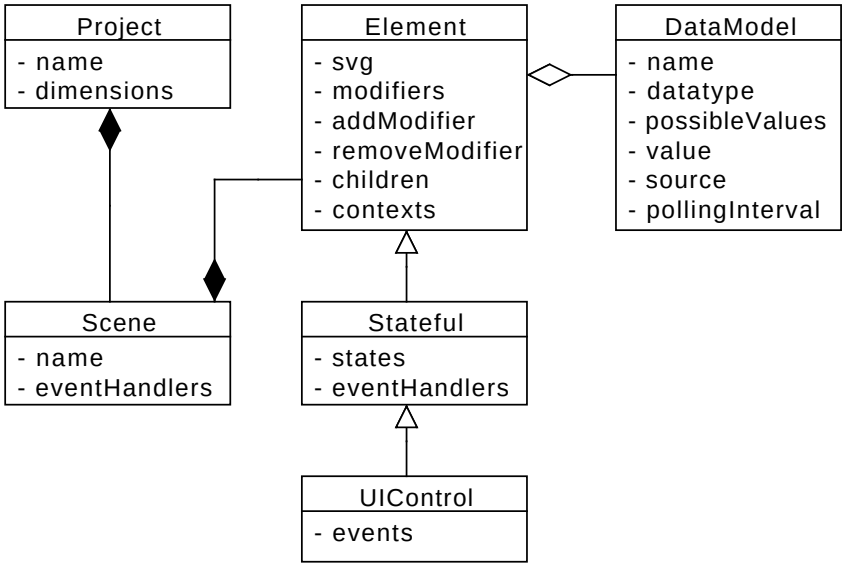


Figure 5.1.: Shows the data models, their most important attributes and their mutual relations.

5.1.1. Projects

Projects contain global information about the project. Currently just a name and canvas dimensions.

5.1.2. Scenes

Scenes are predefined views of the datagraphic and as such consist of Elements. For those elements the scene functions as context. As introduced in Chapter 4, the active context defines,

among other things, which interactions are available. The mapping from events to event handlers is therefore saved in the $eventHandlers$ attribute of the context, in this case the scene.

5.1.3. Elements

First and foremost elements contain the SVG data necessary to display the graphic. Since SVG's have a nested structure, the direct children of this element are saved within the element as well. The $modifiers$ attribute contains the element's modifiers as well as the contexts in which they are active. In addition to the default modifiers there also are $addModifier$ and $removeModifier$ attributes which define how the element is added to or removed from the canvas when this occurs. And lastly, the *contexts* attribute contains all contexts in which some instance of this element is visible (by definition a union of all contexts of all modifiers).

Stateful elements and UI-controls

Some elements can be in different states. In these cases they provide a context for their children. So when the state changes the children's context changes and (among other things) different modifiers might be applied. $Stateful$ elements are just an extension of normal elements. They store the different states in the *states* attribute. Since they function as a context they must also save mappings from events to event handlers.

UI-controls are a special kind of elements that apart from being stateful (see Section 4.3) can also emit events. These are saved in the *events* attribute of the UIControl object. They can be mapped to event handlers within this element or other parts of the datagraphic.

5.1.4. DataModels

The obvious attributes of a DataModel are *name*, *datatype* (currently: number, enumeration or text) and *value*. There also is an attribute $possibleValues$, which in the case of an enumeration is a list of exactly that; all possible values. For numbers it contains the minimum and maximum value and for text it is not defined.

Since data often comes from other sources, a DataModel optionally stores an http address in *source*. The polling interval stored in $pollingInterval$ determines the amount of seconds between each request to the source for an updated value.

5.2. Functional viewpoint

Figure 5.2 gives an overview of the application. It consists of six major components; the Backend, Parser and DataPoller on the server, the Editor, Viewer and ViewerState on the client.

5.2.1. Parser

The most important tasks of the parser are to identify the different elements, their instances, and the differences between these instances. The parser can read and interpret vector drawings in the .svg file format [15]. Every uploaded file represents a scene containing graphical elements. For every element the parser encounters it checks whether it already exists in the database. New elements are saved as is. For existing elements the differences between the original instance and the new one are determined and saved in the form of modifiers, see Section 4.2.1.

In order to check whether an element already exists, it needs to be identified. Since elements can have totally different instances, this task proved to be impossible without some guidance of the designer. This is the main reason why elements that have a purpose beyond just being displayed as a static graphic must be given a unique id by the designer.

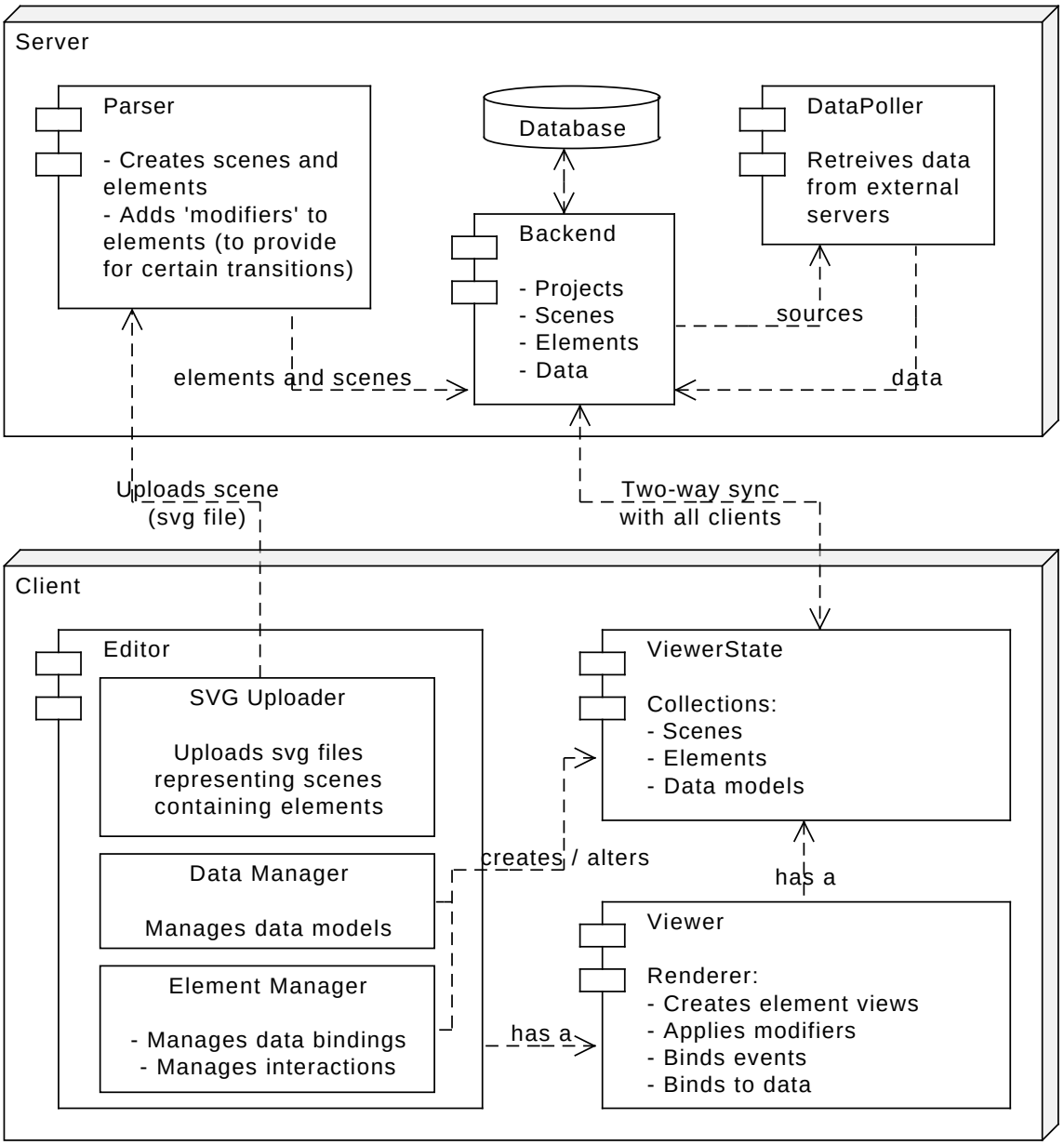


Figure 5.2.: Architecture

Another task of the parser is to identify special kind of elements like user controls or data bindings. Currently, the parser can recognize these elements by a special naming convention of the element ids, provided by the designer. Its implementation is left out as the approach is somewhat debatable. More on this discussion in Chapter 6.

5.2.2. Backend and ViewerState

The Backend is responsible for data persistence and communicating with the clients. In the client the communication is managed by the ViewerState. The ViewerState contains all relevant data in three collections; scenes, elements and data models. The ViewerState acts in the context of the currently active project. When it is first initialized it creates a connection with the Backend on the server and receives the right data. The Backend keeps connections open with

all clients. When the data on the server changes, for instance because the Parser just finished processing a scene, the Backend pushes the new data to all clients currently working on the respective project. And vice versa, when the data in the client changes, for instance when the user has created a new data model, the ViewerState pushes this to the Backend on the server. Because the data on the server now changed all other clients working on the same project will be sent the newly created data model as well. This way the Backend and all the clients are always in sync with each other.

5.2.3. DataPoller

The DataPoller is in charge of collecting data. Data models can have values that are determined by external sources. In these models a polling rate is defined as well. The DataPoller maintains a list of currently active data models (currently in use by at least one client) and polls their values from the external sources at the given rate. These values are passed to the Backend which in turn syncs it with the clients.

5.2.4. Viewer

The main task of the viewer is rendering the datagraphic given a certain ViewerState. Rendering consists of four steps.

- Creating element views;
- Applying modifiers;
- Binding events;
- Binding data.

The whole render process happens by traversing the element tree depth first. This essentially means that elements tell their children to render before they render themselves. This is important since modifiers should effect the children of an element as well. Each render call is given the current context as a parameter. When an element is $stateful$, its children are passed this element as the current context. Otherwise the context remains the same.

When an element is rendered for the first time, an element view is created and placed on the canvas. At this point just the original instance of the element is created. This instance now needs to be transformed to the instance active in the current context. This is done by applying all modifiers of the element associated with this context. In the case the element was already rendered before, a view already existed and whichever instance represented by that view is transformed in the same way to the instance of the current context. The transforming of one instance to the other can happens via a smooth transition. In the future it will be possible to influence these animations as was described in Chapter 3

Since some elements, like buttons, allow user interaction, the right events to make this possible are registered during tree traversal.

When an element is bound to a data model, an event is registered such that the element is re-rendered when data model changes. Databindings were explained earlier in Section 4.4.

5.2.5. Editor

The main task of the Editor is to enable the user to convert .svg files into datagraphics. It has a Viewer to display the datagraphic along the way and a ViewerState which contains all relevant persistent data of the datagraphic. Since the collections in the ViewerState are all that defines a datagraphic it is not surprising that the task of the editor can be divided into the management of these three collections; scenes, elements and data.

Manage Scenes

A scene can be created by uploading a .svg file. Atypically, this doesn't happen via the ViewerState. This is because at this point it is just a file, not the scene and its elements, and therefore not part of the datagraphic yet. The file instead is uploaded by the editor to an upload handler on the server which sends the contents directly to the parser. The Parser then hands the newly created scene and elements off to the Backend which in turn notifies all connected clients including the one that uploaded the graphics in the first place.

Apart from creating scenes, they can be deleted to by altering the ViewerState in the regular way.

Manage Elements

Elements are defined externally in the vector application. In our editor, currently only two things can be managed about an element:

- Events
  When an element is of a type that fires certain events, actions can be registered to these events. A limited set of actions is currently supported as described in Section 4.3.2
- Databindings
  Elements can be bound to data models. This only entails selecting the right data point.

Both these things are saved inside the element itself and synced to the server via the ViewerState.

Manage Data

Data points can be created, edited and deleted in the data manager. In this prototype only a limited set of data types is supported which are described in Section 4.5.

Data points are saved via the ViewerState.

5.3. Concurrency viewpoint

In this section we briefly touch upon the concurrency in the system. Since it concerns a proof of concept, concurrency was not in particular a point of attention. However, a general overview of concurrent processes would be instructive for future developers and testers. We would also like to point out where we experienced difficulties already and where we expect problems might arise in the future concerning this matter.

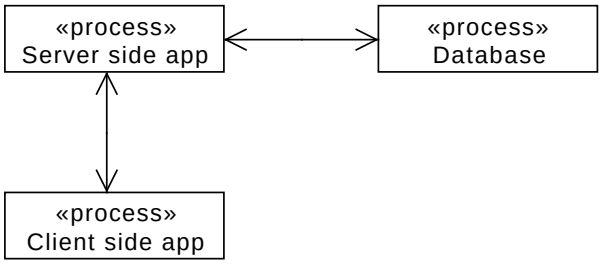


Figure 5.3.: Three main processes run concurrently; the client side application, the server side application and the database.

Both the server side and client side code run in a single-threaded JavaScript environment in which all I/O is non-blocking and event driven. All in all, there are only three processes running concurrently as displayed in Figure 5.3. When for instance a function on the server needs a value from the database, a read event is send to the database and without waiting for an answer execution is continued (non-blocking). When the database is done reading, an event is emitted and if the application has an event handler in place for that particular event, it is executed and passed the requested value. One could say each process exists only of an event loop and event handlers. Since these processes are single-threaded only one event handler can be executed at a time. As a direct result of this platform not much concurrency problems arise. The downside of this *asynchronous* approach is that it requires a very typical event driven implementation in which it can be hard to keep track of execution flow.

5.3.1. Pitfalls

When inexperienced with asynchronous I/O a mistake is easily made and moreover erroneous behavior doesn't always immediately show. For instance, when we started writing the first versions of the parser we produced some code that did two consecutive database calls and expected the first to finish before the second would be executed. This always seemed to be the case but obviously is not ensured in any way. In fact, much later in the process we discovered that seemingly new errors could be explained by this race condition. The only way to make sure the first call finishes before the second is to execute the second in the success event handler of the first. This can become quite inconvenient when nested several levels deep, but fortunately there are great frameworks [1] which considerably ease the pain.

5.3.2. Future considerations

The interprocess communication between the client and the server is an area in which future developers should be wary of concurrency issues. Currently the only feedback loops in place are message acknowledgements. In most cases it is not communicated whether the message was handled successfully or not. This is a problem especially since the communication happens asynchronous and the program continuous execution as if everything is in order. The complications might become even bigger when multiple clients try to edit the same data. In practice we have not encountered any problems in this area yet. Since this is only a proof of concept (of which concurrency is not a focal point) we have not given these issues priority. This is, however, without prejudice to the necessity of a watertight policy when taken into more serious development. Relatively quick progress is expected using more feedback loops in combination with timeouts.

5.4. Development viewpoint

At the beginning of this project we spent a great deal of time and effort to research a set of tools that allow us to maintain solid software engineering principles like testing, continuous integration, encapsulation and dependency management while also having a comfortable development environment. Seeing as how the browser environment is relatively immature and is only recently starting to be used for more complex applications, this was quite a painful process. Nevertheless, we managed to craft a stack that fulfilled most of our requirements and we'll describe it here.

[1] In particular we used Async.js [1]

5.4.1. Development environment

Platform

The client side of the application runs in the web browser. Since we are developing a prototype for a future product, we decided it is acceptable to make use of the latest techniques, even if not all browsers support it yet. We are however only using open standards that are eventually expected to be adapted by all major browsers. In practice, all relevant techniques we use, are part of the HTML5 [14] specification. Google's Chrome is currently one of the leading browsers when it comes to HTML5 support. Therefore we used Chrome to test the application during development.

JavaScript and Coffeescript

A browser environment means having to program in JavaScript, but apart from its functional prowess, there is no denying that Javascript has some shortcomings as a language. Its verbose and sometimes confusing syntax has led to the creation of some languages that compile to JavaScript. One example is Coffeescript[5], a whitespace significant language heavily inspired by Python and Ruby. Even JavaScript's creator and co-founder of Mozilla Brendan Eich has said the following [16]:

> "CoffeeScript is smart and fun new clothing for JavaScript. Clothes are important, and I'd be the first to proclaim that JavaScript needs wardrobe help."

Seeing as our backend also uses JavaScript and we both come from a Python background, this syntax upgrade helped us write cleaner code, faster. Although an extra compile step can be a hassle, we feel that it was worth our while.

Backbone.js, Redis, Node.js and Backbone.io

A necessity for an application like this is a solid MVC framework. We used Backbone, which is the most popular (which means a lot of documentation) and the most flexible. That last part can be a blessing and a curse. To use Backbone effectively a strong knowledge of the DOM model and the jQuery library is required. It doesn't hold your hand and there are multiple ways to do things. Particularly, the event-driven architecture that Backbone encourages can be confusing to fresh eyes. To resolve this we tried to be consistent in our approach and provide solid documentation.

For persistence we used Redis, an in-memory key-value store, that provided us with a speedy way to store our JSON data. Its support for sorted sets was useful for keeping elements in the correct rendering order.

One important aspect of the application is a two-way sync of all project data, between the server and all clients editing or viewing that project. We use Node.js as a server, precisely because it is well suited for this type of real-time application. One other advantage of Backbone comes into play here, since we can use a library called Backbone.IO to sync Backbone models and collections using WebSockets.

Although the async environment can be problematic (as discussed earlier) this type of state syncing can be incredibly powerful and we were very pleased with this setup.

SVG and D3.js

One of the most important aspects of this project is the ability to manipulate SVG elements. The SVG standard[15] has reached widespread adoption among current browsers [4]. The excellent DOM manipulation library D3.js [2]has become a central part of this application. With its focus on data-driven visualizations it is perfectly aligned with our objectives. Extensive

documentation is available and very accessible. We feel we have only scratched the surface of what we can let the datagraphics designer do with this application and D3.js. More on this in Chapter 6

Modularity, package and dependency management

A particular pain point of JavaScript is a lack of a standardized way to create modules. The next version of JavaScript will include module support, but it can take quite some time before it reaches widespread adoption [7]. In the meantime, efforts are being made to provide module support through a 3rd party standard called AMD (Asynchronous Module Definition)[10]. The AMD spec allows for easy dependency management while maintaining asynchronous script loading that is inherent to the web.

We are of the opinion that any complex app should have a way to encapsulate components and define dependencies as to minimize coupling. That's why we used AMD and the Require.js library to achieve these things[17]. While conceptually AMD feels natural, one has to keep in mind that, currently, in a browser environment many HTTP requests are expensive. That's why Require.js provides an optimization build step that compiles the source into one production script, that follows the dependency tree specified[12].

Build tool and package management.

One other important requirement was keeping our build environment consistent. To achieve that on the client side we used package manager that keeps our dependencies on external libraries consistent. A package manager targeted at the browser has been lacking until recently. Engineers at Twitter released Bower[3], a package manager that allows you to specify on which libraries (and versions) your app depends in a component.json file. On the server side NPM (Node package manager) makes defining dependencies easy with the package.json file.

To tie it together we used a build tool called Grunt. It's a very bare bones and transparent way to define tasks that should be part of a custom build. It also has a wide variety of tasks provided by the community such as: compiling to Coffeescript, running tests, optimizing and minifying sourcefiles etc. We have defined several custom builds, but our main build step currently does the following:

1. fetch all external server-side dependencies with NPM as defined in the package.json
2. fetch all external client-side dependencies with Bower as defined in the component.json
3. compile Coffeescript and copy the .js files to a mirrored directory structure
4. compile Hogan.js templates and copy the .js files to the appropriate folder
5. generate documentation from comments in the code using YUIdoc

It's mostly compiling and keeping the environment consistent, because we're not too concerned with optimizations like the ones mentioned earlier just yet. Adding these steps to the build process is a no-brainer, though.

5.4.2. Testing and developer workflow

Developer workflow

For a version control system (VCS) we used GitHub, which uses git and provides an intuitive web interface that allows developers to discuss their code. Our workflow was facilitated by GitHubs "pull request" interface. We made sure our master branch was always stable, at least as stable as a prototype can be, with all tests passing. Whenever one of us was starting work on

a new feature we branched from master, named the branch after the feature and started working on it. When the feature was completed we issued a pull request, telling the other we would like to merge this feature into master and it needs review. The other could ask for improvements by commenting on certain lines of code or merge it in. This workflow was a pleasant way to keep the master branch sane and keep tabs on what the other was working on.

Testplan

Since this is a prototype, getting 100% test coverage didn't get the highest priority. We are, however, charmed by the test driven development (TDD) workflow and strived to implement it in this project. We firmly believe tests can aid development in numerous ways. However, the problem we faced time and time again during the course of this project is lack of time. And there is no denying that writing meaningful tests can be a huge time sink. We quickly found that for every test, there is a judgment call that has to be made: "Is it worth the time investment (stubbing, defining test cases) to properly test this, so that I can save time later on". You'll notice there is no mention of robustness or stability there, and if this project were to leave the prototype phase that focus would obviously have to shift. Of course, we made sure crucial parts get a 100% coverage with all edge cases covered.

Test tools and continuous integration

Our test framework of choice is Mocha[8], which has support for asynchronous tests and also provides a BDD (Behavior Drive Development) syntax. The BDD way of writing tests gives a useful overview of what each test does and we would recommend to keep using it. For mocking and stubbing we used a library called SinonJS[13]. Having to stub or mock certain objects, made us think about applying the "dependency injection" pattern [6] more, which resulted in more flexible code.

From the outset we intended to have some sort of continuous integration (CI) setup. We managed to get one up and running with the help of the most popular CI server called Jenkins. Our main objective was getting to a point where, whenever we pushed to one of our branches, Jenkins would do a build, run the tests and show test/coverage reports and documentation in one place.

Getting test coverage up and running was a very hairy process, since coverage reporting doesn't seem like a commonplace practice yet in the JavaScript world. We had to use a poorly maintained library called JScoverage to instrument our code, then run the tests in a headless browser called Phantom. To get the report, however, we had to jump through some hoops to get our JScoverage output to work with a Jenkins plugin intended for Java coverage called Cobertura (we had to rewrite an adapter for another test framework).

After getting it all up and running these are the steps Jenkins executes on each push:

1. execute Grunt build
2. run tests in PhantomJs
3. generate test and coverage reports with Cobertura

In practice, we noticed that with a 2-man team, consistent build environments and a lacking focus on coverage, the Jenkins server added very little value since we could do all these things locally. We don't believe our efforts have been a waste, however, because now that the basic plumbing is in place, the server can be extended more towards Continuous Deployment by adding more build steps. If the project gets out of prototype phase and/or gets a bigger team, this CI setup will provide useful indeed.

6. Recommendations

Now that we have made the first serious attempt to develop a datagraphics application, we would like to make some recommendations on how to proceed from here on. Although the general concept seems feasible (see Chapter 8) there still are some serious practical issues that need to be addressed.

First of all, we have not been able to come up with a convincing work flow integrating the creation of vector graphics and datagraphics. Creating datagraphics inherently involves adding meta data to the vector graphics. For example, stating that a certain rectangle is a button. But also, since an element can have many different instances throughout the datagraphic (Section 4.2), these instances need to be identified as the same element. The challenge lies in providing an acceptable way to add this meta data. Since we cannot influence the capabilities of current programs we first explored the possibilities of doing this in the datagraphic application. Unfortunately, this proved quite complex and not user friendly. We then realized designers already need to have a clear understanding of these abstract notions in the datagraphic while creating the static graphics. In other words, they already know a certain rectangle is going to be a button when drawing the rectangle. This further convinced us that the necessary meta data is best inputted at that moment. Unfortunately, the only capability graphic programs provide to add extra information to an element is through its id. Using a specific format, this is how all meta data currently needs to be inputted. Essentially misusing the id of elements. Clearly, this solution is far from ideal. Before any further efforts are made in this project, we recommend investigating this issue. To the best of our understanding, there are at least the following scenarios:

- Although not ideal, the target audience finds the current solution acceptable.
  In this case a lot of improvements are still required. In particular the user needs to receive detailed feedback when uploading a SVG file. Are there syntax errors? Are all required states provided for the special elements?

- The current solution is unacceptable and users insist on adding meta data in the graphic environment.
  In this case there are two ways forward:
  - Incorporate a vector graphic editor in the datagraphics application;
  - Enter into a partnership so that current vector graphic programs allow for easy meta data input.

- The current solution is unacceptable, but it there is a way to manage all complexity in the datagraphic application.
  Earlier efforts to achieve this failed, but it might be worth while to look into again. The biggest problem we ran into was the identification of different instances of the same element. This seems impossible without the guidance of the user due to the SVG properties. See also Section 5.2.4. It might be conceivable to have a work flow in which the user can visually select all instances and map them to the same element that way.

- The current solution is not user friendly enough for most infographic designers, but there is a different or smaller target audiance still interested in the product.
  This could for example be the case if one or two people from Schwandt Infographics accept

the steeper learning curve and make the most of it. It is also imaginable programmers already capable of creating datagraphics like to use it to get around some of the heavy lifting. The improvements suggested in the first scenario apply here as well.

To find out which of these scenarios is most plausible, further investigation is required.

Another thing we recommend investigating is how programmers can be integrated into the work flow. In many real world use-cases, some visualizations are too complex for designers to implement and a software developer is required to step in. This could be realized using a widget API as we originally planned to implement, but had to forego due to time constraints. However, it would be indispensable when the application gets taken into production if we want to deliver a more versatile product.

7. Process

This was our first time completing a project using agile development principles for an actual client. We will describe the process and reflect on it here.

7.1. Scrum

As discussed in the orientation report we decided against the traditional waterfall model of extensive requirements gathering and have a dynamic, constantly updating user story/feature list instead. We picked the Scrum methodology because it aligned very well with the experimental nature of the project.

We used a web application called Planbox [11] to track our progress. We filled our backlog with the user stories as defined in Chapter 3 and decided on using 2-week sprints. In what some would call non-agile fashion we made a rough planning of what user stories we would implement in what sprint. Then, at the beginning of each sprint we would start defining the user stories in a much higher level of granularity. We could then define tangible tasks, determine an estimate for the time required for each task and plan the coming sprint accordingly.

Each sprint we met with Dr. Pinzger, our TU supervisor, to discuss our process and he would give us new insights into the agile way of doing things. Each week we would meet up with Mr. Akkerman, our supervisor from Attingo, to discuss our progress and/or setbacks, so we could adjust our expectations and planning. We tried to always have a new version of the prototype to demo at each meeting.

Especially in earlier sprints we noticed this project was very much evolving as we gained new insights about different aspects of the application. As a result, our non-agile rough planning turned out to be quite inaccurate. It quickly dawned on us the prototype would have to be streamlined and some features cut. What you see outlined in this report is that streamlined product and we feel it delivers as a proof of concept, since many insights about the subject matter were gained as discussed in the previous chapter.

Overall, we made an utmost effort to deliver on what we promised and communicate our setbacks to the client.

7.2. Reflection

By far, the most challenging aspect of this project was making an accurate prediction of the time required for a certain task. Since this is relatively unfamiliar territory for the both of us, there were many unscheduled holdups. Getting to know new libraries and tools, fixing bugs, discussions about software architecture, discussions about UI and product philosophy; it all takes more time than we anticipated. We learned that adapting to a new environment results in lots of uncertainty, so one should plan accordingly. To say we bit off more than we could chew, while true, would be easy. We prefer to believe this was an excellent learning experience in time management, and we hope we will gradually get better at it.

This project, besides a great experience in terms of the process, also was a perfect opportunity to improve our programming and software design skills. We had to start from scratch on all fronts. From requirement gathering to selecting the right technologies, to designing a sound architecture and finally implementing the prototype. There were no stepping stones whatsoever. This proved to be a big challenge and in the end both very instructional and satisfactory.

8. Conclusion

The proof of concept developed during this project has convincingly demonstrated the web environment (without plugins) is very well suited for interactive, data-driven visualizations. The concepts and solutions described in Chapter 4 form a sound basis for it. By reducing the complexities of a datagraphic to a state and event based web application, we were able to create a path from static vector graphics to a datagraphic. The challenge lies in providing a user friendly work flow to tread this path. This is what we belief to be the Achilles heel of the project and as described in Chapter 6, we recommend investigating different scenarios in this matter before proceeding on other fronts.

All in all, it can be concluded a datagraphics application in and for the modern web environment is feasible and deserves to be invested into further.

Bibliography

[1] *Async.js*. URL: https://github.com/caolan/async.

[2] Michael Bostock. *Data-Driven Documents*. 2012. URL: http://d3js.org.

[3] *Bower*. URL: http://twitter.github.com/bower/.

[4] *Can I use?* URL: http://caniuse.com/svg.

[5] *CoffeeScript*. URL: http://coffeescript.org/.

[6] *Dependency injection*. URL: http://en.wikipedia.org/wiki/Dependency_injection.

[7] *Harmony*. URL: http://wiki.ecmascript.org/doku.php?id=harmony:modules.

[8] *Mocha*. URL: http://visionmedia.github.com/mocha/.

[9] Doug Newsom and Jim Haynes. *Public Relations Writing: Form and Style*. 2004.

[10] Addy Osmani. *Writing Modular JavaScript With AMD, CommonJS and ES Harmony*. URL: http://addyosmani.com/writing-modular-js/.

[11] *Planbox*. URL: http://planbox.com.

[12] *REQUIREJS OPTIMIZER*. URL: http://requirejs.org/docs/optimization.html.

[13] *Sinon.JS*. URL: http://sinonjs.org/docs/.

[14] W3C. *HTML5*. Oct. 2012. URL: http://www.w3.org/TR/html5/.

[15] W3C. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Aug. 2011. URL: http://www.w3.org/TR/SVG11/.

[16] *What are disadvantages of using CoffeeScript*. URL: http://www.quora.com/CoffeeScript/What-are-disadvantages-of-using-CoffeeScript.

[17] *WHY AMD?* URL: http://requirejs.org/docs/whyamd.html.

A. Original assignment (in Dutch)

A.1. Project voorstel

A.1.1. Opdrachtgever

Attingo Holding BV is een bedrijf dat onder meer andere bedrijven financiert, ermee samenwerkt of er in deelneemt. Ook het geven van adviezen, het voeren van beheer en bestuur behoort tot de werkzaamheden.

Een van de bedrijven waar Attingo Holding in investeert en advies aan geeft is Schwandt Infographics. Dit bedrijf heeft onder andere door de economische crisis ervaren dat het een kwetsbaar business model heeft dat alleen omzet kan genereren uit declarabele uren. Het bedrijf wil naast haar traditionele model een activiteit ontwikkelen waarmee constantere omzetstromen te realiseren zijn in de vorm van een service model. Schwandt infographics heeft hiervoor een concept bedacht dat past binnen de core business van het bedrijf en voorziet in een steeds toenemende vraag uit de markt. Echter om dit concept te gelde te kunnen maken is een omvangrijke IT investering nodig waar het bedrijf de middelen niet voor heeft.

Attingo Services BV, een werkmaatschappij van Attingo Holding BV heeft van laatstgenoemde opdracht gekregen een prototype te ontwikkelen waarmee de technische haalbaarheid van bovengenoemd concept te toetsen is.

A.1.2. Opdracht

Infographics waren tot voor kort hoofdzakelijk statische weergaven in een fysieke wereld. Echter, Interactieve en data driven infographics in de virtuele wereld (datagraphics) zijn sterk in opkomst. Helaas beschikken grafisch ontwerpers niet over de juiste tools om deze datagraphics te maken. Er zijn altijd IT-ers nodig om het eindproduct te realiseren. Initieel gebeurde dat vooral m.b.v. Adobe Flash, maar door de snelle ontwikkeling van browsers is een groter publiek te bereiken door gebruik te maken van HTML5 en Javascript, echter deze technologien staan nog verder af van de grafisch ontwerper.

Wij zoeken een partij die een werkend prototype kan ontwikkelen dat ontwerpers, met minimale tussenkomst van IT-ers, in staat stelt datagraphics te ontwikkelen. Een tool die de ontwerper in staat stelt zijn/haar ontwerp (vector graphics, doorgaans gemaakt in Adobe Illustrator) om te zetten in een interactieve, data driven webapplicatie. Met data driven wordt bedoeld dat de exacte weergave (grootte, kleur, positie, etc) van graphics (elementen) afhangt van data. Zodra op een later moment de data wijzigt zal de weergave dus automatisch ook wijzigen. Het afhankelijk maken van data noemen we data binding.

De te onwikkelen software zal in ieder geval de volgende functionaliteit moeten bezitten:

Vector graphics Importeren, weergeven, data binding.

UI elementen Er moeten basis control elementen (knoppen e.d.) kunnen worden toegevoegd. Deze moeten ook aan te passen zijn.

Scenes Er moeten verschillende scenes gemaakt kunnen worden. Binnen een scene bestaan bepaalde objecten. In het eindproduct is steeds n scene actief. Het moet dan mogelijk zijn te switchen van scene naar scene en op die manier dus andere informatie in beeld te brengen.

Transities Daar waar eigenschappen van objecten veranderen, moeten korte simpele animaties (transities) kunnen worden toegevoegd. Hetzelfde geldt voor overgangen van scene naar scene.

Widget API (application programming interface) Een widget is bedoeld voor complexere objecten die een ontwerper niet middels de standaard functionaliteit kan realiseren. Een software ontwikkelaar moet betrekkelijk snel een widget kunnen maken die de ontwerper vervolgens kan configureren en als standaard object kan gebruiken. Dit kan bijvoorbeeld een bepaalde herbruikbare grafiek of andere data visualisatie zijn. De ontwerper moet de widges wel zelfstandig kunnen binden aan data en hun eigenschappen (kleuren, afmetingen, titel etc) kunnen aanpassen.

A.1.3. Use case

Een groot en drukbezocht themapark wil middels een mobiele app of website haar publiek realtime informeren over de wachtijden bij de verschillende attracties. Daarnaast moeten bezoekers ook kunnen 'inzoomen' op een attractie om extra informatie te kunnen lezen. Het park stelt de benodigde realtime data beschikbaar via een webbased API.

Hierin willen we voorzien door een aantrekkelijke vector tekening van het park te tonen waarop de attracties zichtbaar zijn. Bij iedere attractie is een staafje afgebeeld dat in hoogte en kleur verschilt afhankelijk van de wachtijd. Als op een atractie wordt geklikt wordt extra informatie getoond. Transities van de ene naar de andere weergave moeten via korte simpele animaties verlopen.

Alhoewel we op zoek zijn naar een product dat daadwerkelijk voor simpele projecten ingezet kan worden, zal het ook fungeren als een 'proof of concept' voor complexere projecten. De focus zal daarbij voornamelijk moeten liggen op het visualisatie aspect (en bijvoorbeeld niet op het beheer van data).