Improving the efficiency of SVG in iTasks using deltas

August 24, 2017

Author:
Dion van de Vooren
s4256468

Supervisor:
Peter Achten

Second reader:
Rinus Plasmeijer
Abstract

The current implementation of the SVG library in iTasks (Graphics.Scalable) is inefficient, since every time the image changes, the whole image is recalculated using JavaScript on the (sometimes lightweight) clients. More elegant would be sending a delta to the client, that can easily be applied to the image, such that the server does the calculating in the much faster language Clean. Furthermore, not the whole image needs to be rendered, but only the parts that have changed. The challenge is, achieving this improvement with little to no changes to the interface of Graphics.Scalable. That is important, such that old code does not have to be rewritten to work with this version, and users do not need to learn anything new.

For this research, Graphics.Scalable is adjusted such that it does just that. Two methods for generating the delta will be introduced. The results are very promising, but not perfect. The faster method of the two needs some assumptions that were not present in the original version, and requires a minor adjustment to the interface.
Preface

In the course "advanced programming", one of the assignments was working with Graphics.Scalable. The assignment was to create a railroad and a train that could "drive" by jumping to adjacent tracks in a grid (figure 1). The train moved when clicking on a button. The image was updated quickly, but there was a notable delay. I got more experienced with Graphics.Scalable during my research internship in early 2017, where I have worked on expanding it with the path element and the gradient attribute.

When looking for a master thesis subject, I came to know that iTasks sends updates to the clients using a delta instead of the whole user interface. Graphics.Scalable however did not use the same trick, which made me curious how much the efficiency could be improved, especially for small changes. Since I was experienced and interested in the library, I decided to improve it for my thesis, by implementing the same principle that iTasks uses: sending only the parts of the image that have changed to the client.

I would like to thank my thesis supervisor Peter Achten. He allowed this paper to be my own, but steered me in the right direction when needed. I would also like to acknowledge Bas Lijnse for the help provided starting this research and helping with iTasks whenever I got stuck.

Figure 1: Trains driving along a grid. The junctions and traffic lights can be controlled, as well as moving any of the trains in the direction it is driving.
1 Introduction

The task oriented programming (TOP) paradigm (Achten et al., 2015) has been developed for creating distributed multi-user applications. The main building blocks are tasks instead of functions. This makes it closer to the real world, where every big task consists of smaller tasks. iTasks (Rinus Plasmeijer, 2007) is an implementation of this paradigm, written in Clean, a pure, lazy functional programming language. With iTasks, applications that require a server that serves multiple clients can be built. The applications are built in a compositional manner, the programmer decides which tasks are executed when, which is not the case for normal Clean code. Tasks always have a value, along with an automatically created user interface (UI), which can be empty. It is possible to run multiple tasks in parallel. iTasks combines the user interfaces of tasks running in parallel to a single UI, which can be showed to the user of the application. The user can access the application using a browser with JavaScript support. Generating the UI is based on types. If a task requires the user to enter a string, a text box will be created. If information of type boolean needs to be shown, a check box will be displayed. This extends to any first-order custom type.

Based on such a user-specified first-order custom type (the model), iTasks can automatically generate interactive scalable vector graphics (SVG) images (Achten et al., 2014), provided that the programmer specified how the image should look like based on that model. These images are needed for example to create games, to implement a more attractive UI or for communication. For instance, the model for an image that generates a certain number of circles (as in section 4.2) is an integer, corresponding to the number of circles. Based on that integer, the code specifies how many times the circle should be repeated. iTasks generates the SVG code (Ferraiolo et al., 2000) needed to display the image every time the integer changes.

The programmer specifies images in a compositional way and only with pure functions, in line with task oriented programming. Using Graphics.Scalable,
the positions of all elements contained in an image are specified relative to
each other, unlike in the generated SVG code. Images can be aligned in a
grid, overlay, collage, or above and beside each other. The user-specified
image, together with the model is then being processed, calculating the ab-
solute positions that need to be known in order to generate the code. The
size of the image is therefor not known beforehand.

In iTasks, every part of the UI is bound to a task. If the value of the
task changes, only the part of the UI belonging to that task changes. For
efficiency, not the whole UI is resent if one value changes, but only the part
of the UI that has changed. If, for example, you create a big application with
a running clock showing hours, minutes and seconds on top of the page, not
the whole UI is regenerated and resent. Instead, the client only receives the
string representing the time, and the place where it belongs, in our case on
top of the page in a text field. JavaScript functions running on the client
make sure that field is updated with the correct value, and does not touch
the rest of the application. Obviously, it would not be efficient to send the
whole UI belonging to the application, every time a second passes and the
clock needs to be updated.

Graphics.Scalable does not work as elegant. If the model of an image
changes, the server only sends that model to the clients, and they know how to
convert the model into an image. This is due to the fact that Clean code can
be converted to JavaScript code. The Clean code written by the programmer,
responsible for generating the SVG code, is converted to JavaScript and sent
to the client. This raises two problems that slow down the generation of
the image. The first problem is that (automatically generated) JavaScript
code is much slower than Clean code, and the second is that the client is not
always a powerful device. Many modern lightweight devices are able to run
a JavaScript compatible browser, such as tablets, mobile phones and even
televisions.

Generating SVG code from a user-defined image requires some compu-
tations. One of those tasks is computing the spans and bounding boxes of every element. Since it is required to define all image elements relative to each other, and SVG does not support that, the absolute position in the total image needs to be calculated, and for that the span of every element is needed. Furthermore the span of the complete image is needed. Apart from the span, transformations such as rotating and scaling need to be applied, as well as the interactivity and masks. Lastly, the types that are used by the programmer can not be converted into SVG code straightforwardly. This holds for the elements as well as for the attributes of each element. It could be much more efficient to compute the image on the server, and send some type similar to SVG code to the clients. This however has a negative impact on the bandwidth used, since instead of only the model of the image, all SVG code would need to be sent.

In this research, a solution to this problem is presented. In order to save bandwidth, only the elements of the image that have changed are being sent to the client, similar to how iTasks handles the UI. This can only work, if the changes that are made to the image are small. This is the case for many applications. Letting a train drive along tracks for example, only replaces 2 elements of a grid. One to remove a train being on a track, and one that adds a train on a track (similar to the example shown in section 4.4). For the card game Ligretto, the image changes if a card is played. That card is removed from the hand, and placed somewhere else, which translates to two changes in the image. Ligretto has been used as a case study in previous research (Achten et al., 2014). Another game used in a case study is Trax (Achten, 2013), a game where the goal is to create a closed loop by placing tiles. When a tile is placed, again only a small part of the image is affected, resulting in a small delta. For graphical user interfaces, it is not desired to change everything, but for example only the colour of an element that is highlighted. This solution sends, instead of a model, a tree-structured type, that contains every change to the image since it was last updated. It has
this structure, because SVG code itself is based on XML, which also has a tree structure. If some branch of the tree has not changed, the client will find nothing in the position of the corresponding delta-tree for that branch.

The challenge is generating those changes. Two techniques will be presented. The first one compares the previous and the new processed images, resulting in the difference between them. This implementation is straightforward, but should be done more efficient, since besides generating an image, the server needs to compare images. The second technique combines the creation of the new unprocessed image and the delta, which can be used to send to the client. This however is quite difficult, since processing the changed elements has to be done while constructing the new image, where before they were different steps.

In this research it is investigated whether this method of updating an image is indeed faster than the original, and if yes, how much faster. Furthermore it is discussed if the second technique is more efficient than the first one, and if it is worth the effort of converting the library to generate a delta while constructing the new image.

In the next section (section 2) iTasks, Graphics.Scalable and SVG will be further discussed. In section 3 the version that works with sending a delta instead of a model will be explained, with in section 4 the result, how well it performs compared to the original version. Related work will be discussed in section 5 with the conclusion in the last section.
2 Preliminaries

2.1 Task Oriented Programming

The Task Oriented Programming paradigm has been developed for creating multi-user applications. It is close to daily life, where tasks have to be performed, because it uses tasks instead of functions as a central concept. iTasks is an implementation of the Task Oriented Programming paradigm, embedded in Clean. Clean is a general-purpose pure functional programming language, developed by the Software Technology Research Group of the Radboud University Nijmegen. With iTasks, you are able to create interactive, distributed, multi-user applications without the hassle of handling communication and user interfaces. Being an implementation of TOP, iTasks makes use of tasks that are used in a compositional manner. Web pages are generated automatically, based on types. iTasks contains extensions such as Google Maps and SVG images. The Clean programming environment with iTasks can be downloaded from http://clean.cs.ru.nl/Clean.

An example of a task that asks the user for a phone number, which can be viewed afterwards, is displayed in figure 2.

```clean
task :: Task [Int]
task = getNumber >>= \n -> viewInformation "Phone number:" [ViewAs readable] n

getNumber :: Task [Int]
getNumber = enterInformation "Free editor" [] >>= \n -> n

readable :: [Int] -> String
readable [] = ""
readable [i:r] = toString i ++ readable r
```

Figure 2: The code for a task that asks the user for a phone number.

The task "task" makes sure that the user enters a phone number and that number is displayed. First, the task asking the user for a phone number
is called (getNumber). Since the type of getNumber is ”Task [Int]”, iTasks automatically asks the user for a list of integers. As soon as that task is completed, the main task uses that list to display it. Using the function ”readable”, the list is converted to a string looking like a phone number. The generated web page, in which the user is asked for a list of integers is shown in figure 3.

Figure 3: The user interface that is automatically generated from the code of figure 2. A user can enter a phone number as a list of integers, and view it as a string after pressing Continue.
2.2 How it works

Every task has a task value, which can vary over time. Such a task value can either be no value (if the user didn’t enter anything) or a value that is stable (the user pressed the button and can’t edit the value anymore) or unstable (the user can still edit the value). Tasks have a (part of a) user interface (UI) along with their task value. The UI can be empty for tasks that do not require user interaction. A UI is sent over the network with HTTP, using a single server serving multiple clients. Those clients can access the server with an internet browser that supports JavaScript.

Tasks are compositional, which means they can be combined, using the step operator (\(\gg\gg\)). For example, task1 \(\gg\gg\) \(r\). task2 \(r\) means, that first task1 is executed, and then task2, with the task value of task1 as argument. Furthermore, tasks can be executed in parallel, using -||-. task1 -||- task2 makes sure that both tasks are running at the same time, the result of that combination is the result of the task that delivers a stable value first. (task1 -||- task2) is a task itself, and because every task generates a (possibly empty) UI, -||- needs to combine the interfaces of both tasks. There are alternatives for \(\gg\gg\) and -||-, which work slightly different, but the basic function is the same. In the end, all interactive tasks are derived from the core task interact.

```plaintext
interact :: d EditMode (RWShared () r w)
  (r -\(\rightarrow\) (l, v))  //On init
  (v 1 v -\(\rightarrow\) (l, v, Maybe (r -\(\rightarrow\) w)))  //On edit
  (r 1 v -\(\rightarrow\) (l, v, Maybe (r -\(\rightarrow\) w)))  //On refresh
  (Maybe (Editor v)) -\(\rightarrow\) Task (l,v) | toPrompt d & iTask l & iTask r &
  iTask v
```

This means, every tasks contains a description d, an EditMode specifying whether the user is viewing, entering or updating information and a reference to shared data the task has access to. Furthermore the behaviour when initialized, edited and refreshed is specified by the corresponding functions. v corresponds to the view of the task, l is the type of the local state. r and
w are types that can be read and written respectively. There is room for a custom Editor, changing the look and feel of that task in the application. If the Editor is not specified, the default Editor will be used. The Editor has the following type:

```haskell
:: Editor a =
  { genUI :: DataPath a ∗VSt −→ *(MaybeErrorString (UI, EditMask), ∗VSt)
  , onEdit :: DataPath (DataPath,JSONNode) a EditMask ∗VSt −→
    *(MaybeErrorString (UIChange, EditMask), a, ∗VSt)
  , onRefresh :: DataPath a a EditMask ∗VSt −→ *(MaybeErrorString (UIChange,
    EditMask), a, ∗VSt)
  }

The functions contained in the Editor will be converted to JavaScript and run on the client. The Editor is the successor of the Editlet (Domoszlai, 2014). Since sending the whole UI every time something changes is very inefficient, the server sends a delta of only the elements that have changed since the last update. The DataPath is used to identify sub structures in a composite structure, a corresponds to the value of the task. The EditMask is used to hold information about a value when it is being edited. UI is the user interface generated by every interactive task. VSt will be further explained in section 3.3. The Editor, running on the client, receives that delta and changes the Document Object Model (DOM) accordingly. The type of the delta that is sent is a tree, defined by:

```haskell
:: UIChange
  = NoChange
  | ReplaceUI UI
  | ChangeUI [UIAttributeChange] [(Int,UIChildChange)]

:: UIAttributeChange = SetAttribute !String !JSONNode

:: UIChildChange
  = ChangeChild !UIChange
  | RemoveChild
```

11
NoChange means that nothing has changed. ReplaceUI sends a whole new UI, which is convenient if many things have changed. The most important one is ChangeUI, which contains two lists. One of type UIAttributeChange used for the changes in the current node, and one for the changes concerning the children of that node. The order of the list has no effect on the resulting UI, because the changes applying to the children are indexed. Children can be removed, inserted, moved and changed (which is again a UIChange). The clients can send back information as well. This is done by listeners that come along with everything the user can interact with, such as text input fields and buttons. Every time the user does something that affects a value in the application, a JavaScript function sends that data to the server. The server running the tasks depending on those values can then create a new UIChange and pass it to the clients. A model of this communication is displayed in figure 4.

2.3 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is a vector image format. It uses polygons to represent images, instead of a raster filled with colors. Vector image formats have some advantages over raster based images. It is possible to scale images without loss of quality, and pixelated images as a result of overenlarging will not occur. Also, less data is needed to describe an image. In the case of SVG another advantage is, since SVG is based on XML and thus is readable text, editing an image with a normal text editor goes without much effort. There is however some software available for editing the images, such as Inkscape and Adobe Illustrator.

SVG is a standard, developed by the World Wide Web Consortium (W3C) in 1999. It is supported by all modern major browsers. An SVG image consist
OnUIChange (initialUI)
doEditEvent (taskid, editorid, val)
OnUIChange (change)
Display the UI
Update the state and send to clients
Process the change

Figure 4: A model of the communication between server and client. OnUIChange and doEditEvent are JavaScript functions, executed on the client. The arguments of OnUIChange are generated by tasks on the server, doEditEvent is called when the user enters a value.

of basic shapes, such as circles, rectangles and paths, which can be grouped and transformed. Figure 5 shows an example for some SVG code with the corresponding image.

```xml
<svg height="300" width="500">
  <rect width="300" height="100" style="fill:blue" />
  <ellipse cx="240" cy="120" rx="220" ry="50" style="fill:red" />
  <circle cx="210" cy="70" r="70" style="fill:green" />
</svg>
```
2.4 Graphics.Scalable

The SVG implementation in iTasks (Graphics.Scalable) \cite{(Achten et al., 2014)} does not contain the whole SVG library. It does contain all basic shapes and lines: rectangle, circle, text, oval, line, polyline, polygon and path. Some operations (fill, stroke, opacity, masking and gradient) and transformations (fit, scale, rotate, flip and skew) can be applied. Furthermore images can be made interactive, linking a function to clicking on a part of an image.

Unlike in the SVG library, where all parts of the image are specified in absolute coordinates, in Graphics.Scalable all elements of the image are relative to each other. They can be put in a grid, a collage, overlay. Images can also be put above or beside each other. Going from a grid of images to actual SVG code, where coordinates are absolute, takes up a big part of processing the image. An example of a grid specified in iTasks with the corresponding SVG code and image is shown in figure 6.

An example of using an interactive SVG image as an UI, letting users enter a phone number using an image, can be seen below (figure 8). Next to that image is the default editor for lists of integers and is acquired without hassle, as can be seen in the corresponding iTasks definition (figure 7).

The main task is "task", which makes sure the sub tasks (viewing the
Figure 6: The SVG code and image that is generated from the Clean code, displaying how a grid looks like.
Figure 7: iTasks code to let users enter a phone number using the “free” Editor and using an image. The entered phone number can be viewed afterwards. The UI resulting from this code is displayed in figure 8.

phone number, the default editor and the image) are combined. The task imageTask, which is not displayed, is responsible for displaying the image defined in the function myImage.

2.5 Internal working

If some task value or shared data source changes, that value is sent to every client depending on it. The client, running the Editors belonging to the
Figure 8: The web page rendered from the code in figure 7. At the top the entered phone number can be seen, below that are two ways to enter a number. At the left the automatically generated editor (as in figure 3) and at the right the SVG image, where numbers can be entered by clicking on them.

elements in the application, receives that value and passes it to the correct Editor. If it is an Editor belonging to a text field, the text within that field is replaced with the new value. If however the Editor should generate the SVG image, the whole image needs to be rendered from scratch. A diagram of the communication between server and client is shown in figure 9. During rendering the following tasks have to be performed:

1. Compute span of each element

Since every element of the image needs to be defined relative to each other which is not supported by SVG, the absolute coordinates need
to be calculated. Converting for example an element in a grid to SVG code, involves calculating the span of every element that is before it in the grid. For some elements, such as text fields, calculating the span can only be done on the client.

2. Convert user-friendly type to some type similar to SVG code

Specifying an element in iTasks is quite simple, but does not correspond to SVG code. The image generated by the programmer needs to be converted to a type similar to SVG code. A circle, specified by “circle diameter”, is converted to a circle element, with SVG attribute radius being half of the diameter.

3. Apply default attributes

All basic shapes have default attributes, such as the width and color of the stroke. These attributes need to be applied, in case that the programmer did not specify them.

4. Apply transformations

If transformations were applied to (a part of) the image, they need to be processed. This involves calculations like converting degrees to radians when rotating elements.

5. Handle interactivity

The JavaScript code for interactivity, such as clicking on elements, needs to be generated.

6. Apply masks

If the programmer specified a mask, the span of the element(s) involved need to be adjusted.
Figure 9: The original model of the server-client communication of SVG images. Only the state is sent to the client, where JavaScript makes an image from that state.
3  Delta-based algorithms

With the current implementation, the whole image is regenerated within JavaScript on the client every time the image changes (figure 9). The image is updated using a state. The code needed for the image generation is written in Clean and then converted to JavaScript. Since JavaScript is very slow compared to Clean and the client may be lightweight, such as a mobile phone, it would be much more efficient to just send the things that have changed and let iTasks compute the changes on the server, as it is the case for the UI in iTasks. If for instance a circle is added to an image, not the model of that image is sent. Instead, some type specifying that a circle should be added at a certain position will be sent (figure 10).

In order for this to work, two challenges must be met. The first one is that a new type is needed, that can contain all changes, and is straightforward to convert into SVG code. That would be the type that the server sends to the client. The other challenge is generating those changes.

For the generation of the delta two methods will be discussed. One of them compares the old, and the new processed images, which results in a delta. The other method uses the old unprocessed image in order to generate the delta while constructing the new image. For both methods, saving the old image is necessary. It is important, that the interface of Graphics.Scalable does not change significantly. Previously written code should work, without (or with minor) adjustments on the new version.

3.1  Type of the delta

Instead of sending a model of the image, some type must be created containing every possible change in the image. Furthermore, that type should have a similar structure to SVG code, such that converting is straightforward. SVG is based on XML, which has a tree structure. Updating XML data is not new (Yu et al. 2005), it has been done using the operations insert, delete and
update, all with a path to the affected node. A similar strategy will be used here. The difference is that instead of a set of operations, a tree containing all changes is sent. The type of the delta is:

:: SVGChange
   = Change [(String, String)] [(Int, SVGChange)]
   | Add SVGElt
   | Remove
   | Replace SVGElt

Both methods that generate a delta (discussed later in this section), create an SVGChange. The operations within SVGChange will be applied to the
SVG code. Add appends an element at a given index to the parent node, specified in Change. Remove removes the element at some index. Replace replaces the current element with a different element. Change modifies nodes, in such a way that its attributes and children can be changed. The first list passed with Change contains changes that need to be applied to the current node, corresponding to the attributes. This list is not used in this research, both delta generation algorithms discussed later in this section produce a Replace if the attributes are not equal. It can be used in future work to improve efficiency. The second list contains all changes to the children of that node, where the integer serves as an index specifying the child belonging to the change.

SVGChange is similar to UIChange, discussed in section 2.2. The difference is, that "change" works differently. In the case of SVGChange, a child is of type SVGChange as well, where for UIChange there is a type exclusively for child elements. Furthermore, UIChange supports moving an object to a different position. SVGChange might lead to a more efficient delta with "move" supported, but generating such a delta would be more complex.

### 3.2 Applying the delta

SVGChange has a tree-like structure, with Add, Remove and Replace as leaves, and Change as a node. SVGElt was used in the original version, and can be a simple SVG element, such as a circle, but might also be a group of elements. More details on SVGElt can be found in section 3.4. Using this structure, it is straightforward to apply the delta to the branches of existing SVG code that have changed, without affecting the rest of the code. If some branch of the SVG image has not been changed, its index will not appear in the list of child-changes belonging to Change.

The code needed to apply SVGChange to the existing SVG code on the client is written in Clean and converted to JavaScript. It applies the delta to the top node of the SVG code. Since that node is always the same, a delta
starts with Change. The second list of Change specifies which changes should be applied. The function applying those changes goes through that list, and performs the operation (add, remove, replace or change the sub-tree) at the given index.

### 3.3 Saving the old image

Keeping the previous image on the server is necessary to compute the delta. This needs to be done without changing the functions within the Editor, since it would affect everything within iTasks dealing with an Editor. Every Editor comes with a state of type VSt, keeping track of different properties of the task (section 2.2).

```plaintext
:: *VSt =
  { taskId :: String  // The id of the task
    , mode :: EditMode  // entering, updating or viewing data
    , optional :: Bool  // Create optional form fields
    , selectedConsIndex :: Int  // Index of the selected constructor
    , iworld :: *IWorld  // Contains information about the server
  }
```

Within that state, an IWorld is contained, storing information about the server. For this research, an extra field was added to the IWorld, which acts as a general server cache and can be used for storing any type. This cache will be used for storing the previous image in the methods of sections 3.4 and 3.5.

### 3.4 ”Diff” - Generate the delta by comparing images

When the difference between some object1 and object2 is needed, the obvious idea is to compare them. Having the object1 and the difference between the objects, it is possible to construct object2. This idea has been researched and used in practice. Google Docs is an application where multiple users
can work on the same document at the same time. Similar to keeping the SVG image updated, the challenge is to keep the latency as low as possible. The underlying synchronization mechanism Google Docs uses is differential synchronization \cite{Fraser2009}. The algorithm used for this research is less advanced, but the basic idea is the same.

When the model for an image changes, the image will be processed as discussed before. Simplified, a processed image is a list of SVGElt. It is quite similar to SVG code, which makes the conversion straightforward.

\begin{verbatim}
 :: SVGElt = SVGEl [HtmlAttr] [SVGAttr] [SVGElt]
  | CircleElt [HtmlAttr] [SVGAttr]
  | ClipPathElt [HtmlAttr] [SVGAttr] [SVGElt]
  | DefsElt [HtmlAttr] [SVGAttr] [SVGElt]
  | EllipseElt [HtmlAttr] [SVGAttr]
  | GElt [HtmlAttr] [SVGAttr] [SVGElt]
  | ImageElt [HtmlAttr] [SVGAttr] [SVGElt]
  | LinearGradientElt [HtmlAttr] [SVGAttr] [SVGElt]
  | LineElt [HtmlAttr] [SVGAttr]
  | MarkerElt [HtmlAttr] [SVGAttr] [SVGElt]
  | MaskElt [HtmlAttr] [SVGAttr] [SVGElt]
  | PathElt [HtmlAttr] [SVGAttr]
  | PolygonElt [HtmlAttr] [SVGAttr]
  | PolylineElt [HtmlAttr] [SVGAttr]
  | RectElt [HtmlAttr] [SVGAttr]
  | RadialGradientElt [HtmlAttr] [SVGAttr] [SVGElt]
  | StopElt [HtmlAttr] [SVGAttr]
  | TextElt [HtmlAttr] [SVGAttr] String
  | RawElt String
\end{verbatim}

The previously processed image (if available) is retrieved from memory and the new image is saved. Now that both the old and the new image (both of type [SVGElt]) are available, they can be compared.

Comparison is done by comparing every element in the list. First the data constructor and its attributes are compared. If they are different, a
Replace is generated, with whatever is in that position of the new image. If they are the same, the lists of children are compared. In case the element had no children, or all children are the same, the branch will remain empty, and thus will not be sent to the client. If however one or more children are not equal, a Change is generated, containing every child that has changed. Add and Remove are generated, if the list of children of the new image is longer or shorter respectively.

This method adds some additional steps to the original method. Images need to be saved and loaded, and the delta needs to be computed, serialized and applied. The only thing gained, is that the client does not need to process the image anymore. How much impact this has on the total computation time is discussed in the result section (section 4).

The downside of this way of calculating the deltas, is that not always an optimal delta is generated. If the first child is changed (and has an different span), removed or an element is added in the first position, all children of the parent node might produce a Replace. This happens if the second child if different from the third, the third from the fourth and so on.

3.5 "Fused" - Generate the delta together with the image

Integrating the generation of a delta deeper in the current system, might be a more efficient way to generate a delta than comparing images after processing them. One way to do this, is generating the delta while constructing the image. An image can consist of multiple elements, where in Graphics.Scalable every element is created with a function. The function needed to generate a circle is

\[
\text{circle} :: \text{Span} \rightarrow \text{Image} \ m
\]

where the Span corresponds to the diameter of the circle and m is the model of the image. Detecting a change can be embedded in these kind of functions.
This means, the delta will be generated before processing the image. As a result, only the delta needs to be processed instead of the entire image. All elements of an image are generated by functions of the form

\[(\text{some arguments}) \rightarrow \text{Image } m\]

In order to make those functions generate a delta as well, those functions change to

\[(\text{some arguments}) \rightarrow (\text{Image } m, \text{ImageChanges } m)\]

where \((\text{Maybe } (\text{Image } m))\) corresponds to the previous image, which might not be available. The image in the output of those functions is needed to serve as the old image in the next iteration. \((\text{ImageChanges } m)\) is an intermediate data structure, which will need to be processed (to type SVGChange) before it can be sent to the client.

\[:: \text{ImageChanges } m \]
\[= \text{ImageNoChange} \]
\[| \text{ImageReplace } (\text{Image } m) \]
\[| \text{ImageAdd } (\text{Image } m) \]
\[| \text{ImageRemove} \]
\[| \text{ImageChange } \left[\left(\text{Maybe } (\text{Int, Int, (Real, Real))}\right)\right] \left[\left(\text{Int}, \text{ImageChanges } m\right)\right] \]

This type is similar to SVGChange. ImageNoChange was added, to indicate that that part of the image has not been changed. ImageReplace and ImageAdd contain \((\text{Image } m)\) instead of SVGElt as is the case for SVGChange, because \((\text{Image } m)\) needs to be processed. The first list of ImageChange has a different type than Change contained in SVGChange, the reason for this will be given later in this section (page 28).

Within the functions that generate SVG elements, it is checked whether the previous image is equal to the current element. If it is, an ImageNoChange is generated, otherwise an ImageReplace with the new image. The old code for generating a circle is:
circle :: Span → Image m

= mkImage (Basic CircleImage (d, d)) where
d = maxSpan [zero, diameter]

Using the new definition, it results in:

circle :: Span (Maybe (Image m)) → (Image m, ImageChanges m)
circle diameter maybeOldImage
  = (image, changes) where
d = maxSpan [zero, diameter]
image = (mkImage (Basic CircleImage (d, d)))
changes = case maybeOldImage of
  Nothing = ImageAdd image
  Just oldImage = case oldImage.content of
    (Basic CircleImage (span1, span2)) = if (span1 == d && span2 ==
      d) ImageNoChange (ImageReplace image)
    _ = ImageReplace image

If for example, the circle function is called, and the previous image was
a rectangle, it results in an ImageReplace with the image of a circle. It is
however not desired if the programmer needs to pass the old image as an
argument, but only wants to specify the diameter argument. This is why the
function rendering images was changed to take the function

(Maybe (Image m)) → (Image m, ImageChanges m)

instead of just (Image m) as an argument. For the programmer nothing
changes, since calling circle with a diameter results in the requested function.

Combining images is where the difficulties lie. Instead of requesting a list
of images, those functions must now request a list of functions

(((Maybe (Image m)) → (Image m, ImageChanges m))

generating an image and a delta. The grid for example, was a function
but had to be adjusted to

\[(\text{GridDimension GridLayout [XYAlign] [ImageOffset] [(Maybe (Image m))] \to (Image m, ImageChanges m)} \to (\text{Host m}) \text{ (Maybe (Image m))} \to (\text{Image m, ImageChanges m}))\]

With this solution, nothing changes for the programmer. Within the function, it is checked whether the previous image was a grid as well. If that is the case, the elements of the grid are extracted from the old image, and given to the list of functions in the correct order. This results in a list of (Image m, ImageChanges m). If every (ImageChanges m) of that list is an ImageNoChange, the whole grid is the same, and returns a ImageNoChange. Otherwise, an ImageChange is generated, with in the second list every element that has changed, and in the first list the corresponding index of that element, the row size of the grid, and the size of the element. This is needed for calculating the absolute coordinates of that element, which would otherwise be calculated when processing the image. This solution is only usable for some images, since it assumes that all elements of the grid have the same size, which is often not the case. Eliminating this assumption requires rewriting Graphics.Scalable. The other functions for aligning multiple elements have been done in a similar way.

There is one more challenge to meet: transformations and attributes. They are added using functions defined by

\[(\text{some arguments}) \text{ (Image m)} \to \text{Image m}\]

but needs to be changed to

\[(\text{some arguments}) \text{ ((Maybe (Image m))} \to (\text{Image m, ImageChanges m})) \text{ (Maybe (Image m))} \to (\text{Image m, ImageChanges m})\]
However, such a function is not possible, since it is applied after the function generating the delta is created. In order to create that delta however, it must be known which operations have been applied, otherwise it can not be compared to the previous image (with all operations already applied). To make things more complicated, there are default attributes that are applied if that attribute was not specified by the user, such as stroke color and width.

The solution is adjusting the function every element returns, from

\[
\text{(Maybe (Image m))} \rightarrow \text{(Image m, ImageChanges m)}
\]

to

\[
\text{(AppliedTransformations m) (Maybe (Image m))} \rightarrow \text{(Image m, ImageChanges m)}
\]

where \(\text{(AppliedTransformations m)}\) is defined by

\[
:: \text{AppliedTransformations m} ::= \text{(Set (ImageAttr m), [ImageTransform], [SVGGradient, String])}
\]

This type can contain any transformation and attribute that can be applied to the images.

The type of the functions adding transformations and attributes becomes:

\[
\text{(some arguments)} \ ((\text{AppliedTransformations m}) \ (\text{Maybe (Image m)}) \rightarrow \text{(Image m, ImageChanges m)})
\]

These functions only apply the arguments \(\text{(AppliedTransformations m)}\) and \(\text{(Maybe (Image m))}\) to the function, with the corresponding transformation added to \(\text{(AppliedTransformations m)}\). Elements, such as circles, now receive the previous image and all transformations that have been applied to the current image. With that information the delta can be computed. The default attributes are also no longer a problem, since the functions creating the elements can apply them if they are not contained in \(\text{(AppliedTransformations m)}\).
With these adjustments to the library, nothing has changed for the programmer, except that every image now has a function as a type, instead of just (Image m). With this approach, the delta is equal to the one obtained when comparing two processed images. As a result, the same downside occurs, that when the first element of a node is added or removed (and every child is different), only Replaces will be generated.

For this research, not every processing step regarding the delta is implemented, so not every image is possible. An other limitation is that a composition in a composition is not supported. A grid inside a grid for instance is not always updated correctly when using ”Fused”. An overview of what is supported:

1. Compute span of each element
   The span of every element in for example a grid needs to be equal.

2. Convert user-friendly type to some type similar to SVG code
   Supported

3. Apply default attributes
   Supported

4. Apply transformations
   Supported

5. Handle interactivity
   Not supported

6. Apply masks
   Not supported

For the remaining processing steps a lot of coding has yet to be done, especially removing the assumption that all spans in a grid need to be equal
will be a lot of work. Whether or not it will be worth the hassle compared to the other approach will be discussed in the next section.

3.6 Discussion

Which of the two algorithms generates a better, smaller delta, and thus can be more efficient has not been discussed. This is because the generated delta is (apart from rare cases) the same. If the image consists of one element that changes, both methods will generated a Replace. The only difference here is, that "Diff" compares the element after, and "Fused" before it has been processed. An image in Graphics.Scalable can only consist of multiple elements if they are placed relative to each other, in a composition. "Fused" compares the elements of a composition in the order they were added. If the old image is a composition with elements \([e_1, e_2]\) and the new one contains \([e_3, e_4]\), first \(e_1\) and \(e_3\) are compared, then \(e_2\) and \(e_4\).

Processed images, compared by "Diff", also consist of a list of elements ([SVGElement], see section 3.4). These lists are compared in the same way. The delta generated by both methods is equal, if elements maintain their position in the list while processing, which is the case.

In some rare cases however, a completely different delta is created. This happens, if a programmer creates the same image twice, but in a different composition. Before processing they would be different, since a different composition is used, but after processing all elements and attributes are equal. In this case "Diff" would generate no change, and "Fused" would generate a Replace with the entire composition. An example for this is a grid with some elements, that changes into a collage with the same elements, and the positions set to the exact same positions as the elements in the grid (such that it looks like a grid).
4 Results

4.1 Benchmark

There are 2 important aspects when improving the efficiency of a software system, the computation time and the memory space that is needed. For this research, I will only focus on computation time, since space is not an issue for a server and clients with this kind of programs. The original version and the two methods for generating a delta are benchmarked 10 times per image, the results in this section are averages. The complete results can be found in the appendix.

The computation time can be divided in three parts:

- Computing the delta on the server
  Computing the delta will be benchmarked by measuring the time between sending the new model to the server and receiving the delta.

- Communication (size of the message)
  The time the message needs to go from server to client is related to its size, so the size of the message will be measured. This is more general, since connection speed, busy channels and other bottlenecks unrelated to this research are not taken into account.

- Rendering the new image on the client
  The rendering of the image is always done by a browser. All tests are performed in Google Chrome version 60.0.3112.78 (64-bits), running on Windows 10. Google Chrome has a built-in tool for monitoring the duration of the different actions the browser has to perform, such as scripting and painting.
4.2 2000 circles, increment 1

This image consists of a grid, with small circles in it. The amount of circles can be specified on the generated web page. For this test we start with 2000 circles, and measure the time it takes to add one circle. Since this test is performed 10 times, it results in an image of 2010 circles. Figure 11 shows what it looks like when it is fully rendered.

Figure 11: 2010 red circles on a blue background.

For ”Fused”, it is important that all elements are of the same size, which is the case for this image. Furthermore, since circles are only added and removed at the end of the grid, this is an optimal image for this problem.

For tiny changes, as is the case here, the performance difference is huge (figure 12). ”Diff” is 6.4 times as fast as the original. ”Fused” performs even better, using only a ninth of the total time needed in the old version. Notable is that ”Fused”, generating the delta and the image simultaneously, is as fast as generating just the image. In line with the expectations, comparing two images after the new image is processed as is the case for ”Diff” takes longer than just processing the image.
Figure 12: The time between sending a new value to the server and displaying the new image, divided into delta generation (server) and processing the delta (client). This result belongs to the image with 2000 circles, measuring how long adding one circle takes.

The difference in client time is mainly responsible for the resulting total time. With the original version, it took more than a second to convert the model into an image. Applying the received delta on the other hand takes only 8-9 milliseconds.

For a system like iTasks, made for users to perform fast, maintaining attention is crucial. Earlier research (Nielsen, 1993) showed that some application feels like it responds instantaneously, if it responds within 0.1 second. After 1 second, the users flow of thought is interrupted. The old system, needing more than 1.1 second to respond to a new value, exceeds that threshold. Both methods creating a delta do not take more than 0.2 seconds, where "Fused" comes close to feeling instantaneously, with total time of 127 milliseconds. For small images (>200 elements), this difference is not that important, since the total response time would be close to 100 milliseconds.
Figure 13: The amount of characters that is being sent when adding one circle to the image.

Depending on the connection between server and client, the amount of data being sent with each change might be important. As expected, the original version sends an unbeatable 4 characters to the client, being the integer defining how many circles should be generated (a four digit number, 2001-2010). Since both delta generation algorithms output the same delta, its size is equal. For both, the size is 409 characters (figure 13). This is a lot higher than the 4 of the original version, but the delta contains the type of element, its size, color and so on. 409 characters sounds like a lot compared to 4, but is actually only 409 bytes. As a comparison, the global average internet speed in Q1 of 2017 was 900000 bytes per second (Akamai 2017).
4.3 1000 circles, increment 100

For this test, the same image is used (figure 11), but instead of incrementing 1 circle, 100 circles are added with every iteration. This represents a much bigger change to an image. Instead of starting with 2000 circles, now we start with 1000. This is because browsers struggle with the JavaScript code involved with more than 2000-3000 elements, resulting in an empty web page.

Figure 14: The time between sending a new value to the server and displaying the new image, divided into delta generation (server) and processing the delta (client). This result belongs to the image with 1000 circles, measuring how long adding 100 circles takes.

For bigger changes, the difference in performance is less impressive (figure 14). Both versions using a delta are roughly 2 times as fast as the original, where ”Diff” is slower than ”Fused”. The computing time required to generate an image or delta on the server is in line with the previous test. For the original version, processing the image took less time, since (on average) the image contains less elements. The time needed to process the delta however
changed significantly, going from under 10 ms to about 170 ms. This makes sense, since the client needed to apply a much bigger delta (100 changes versus 1 change). It seems that after a certain amount of changes, the versions sending a delta become slower than the original.

Figure 15: The amount of characters that is being sent when adding 100 circles to the image.

As displayed in figure 15, the size of the data that needs to be sent to the client grows drastically, to almost 40000 characters. For devices with a slow bandwidth, this might be an obstacle (about 40 kB). The size of the packets is the reason that the client struggles with big deltas. The delta is converted to the JSON format in order to be sent to the client, and the client needs to unpack it. This is where the most computation time on the client is spent on.

Sending a delta instead of a model to the client is not a perfect solution, since it is not efficient if big parts of an image change. It can however be used side by side with the original version. The amount of changes can be
counted. If it stays below a certain threshold, the delta can be sent, otherwise the model of the image.

4.4 Moving along a grid

This example represents something moving along a grid (figure 16). It is related to the train example in figure 1, which is not tested because “Fused” is not able to render the image (the train and tracks are compositions, placed in a grid).

All 1000 elements of the grid are different, which is closer to real-world examples than the examples from section 4.2 and 4.3. This image is benchmarked, by letting the yellow circle start at the position shown in the image, and measuring the time and data needed to move it one position to the right.

![Figure 16: A yellow circle moving in a grid of 1000 elements, where all elements are different.](image)

The result shown in figure 17 is in line with previous results. It is not as impressive as the result from section 4.2 but that is because the image is about twice as small. That reduces the time the original method needs to construct the whole image, while the time needed to apply a delta increased a little (applying two changes instead of one).
Figure 17: The time between sending a new value to the server and displaying the new image, divided into delta generation (server) and processing the delta (client). This result belongs to the moving along a grid image.

In all results can be seen that the original version and "Fused" both use almost the same amount of computation time on the server. This means, the delta can be created at little to no cost. Since "Fused" creates the image that the original version returns along with the delta, the main argument not to compute deltas (the extra computation time) is discarded.

As expected, the amount of characters being sent from server to client is a lot higher when using a delta (figure 18), similar to previous results.
5 Related work

The challenge of keeping some external data synchronized, with as little latency as possible, is common. Some research about this has been done. The most general one describes synchronization using RDF (Resource Description Framework) graphs [Berners-Lee and Connolly 2004]. RDF graphs however do not have a tree structure, making it unusable for this research.

The problem of recalculating the whole output for some small change in input is similar to the one tackled by "Fused" but has been solved using a different approach [Acar and Ley-Wild 2008]. In this research, programmers need to adjust their code to make it a self-adjusting program. The input goes to a Mutator first, which generates the delta of the input. The delta is passed to the program, where it can be used to only recalculate the parts that have a different output. It differs from "Fused" in the sense that it takes the old input to generate a delta, where "Fused" takes the old output which
is propagated through the program. The goal of ”Fused” is decreasing the computation time needed to create a delta between the old and new output, by omitting comparing both outputs.

The inspiration for this way of changing an SVG image however came from iTasks (Rinus Plasmeijer, 2007), where the user interface is updated in a similar way (as shown in section 2.2). Google Docs also makes use of sending a delta instead of the whole document, the delta is generated by comparing the old and the new document (Fraser, 2009). Just like for SVG images, a low latency is crucial. This method however, which is asynchronous and highly scalable, is too advanced for graphics.scalable.

Synchronizing a calendar or contacts with a server usually works similarly, but latency is not really an issue. Related research includes cloud synchronization (Li et al., 2013), where a low bandwidth usage is the challenge. SVG is XML based, keeping XML code in sync has been done before (Yu et al., 2005). In that research updating an XML file is discussed, using a different strategy than the iTasks implementation does (as discussed in section 3.1).
6 Conclusion

Generating a delta on the server and sending it to the client is much more efficient than sending the model of an image, and letting the client generate the image in some cases. The rule of thumb is, the smaller the delta, the more efficient it is to send the delta instead of the model. When using this approach, it needs to be considered that if the delta is too large, it might be slower than sending the model of the image. An optimal solution would be counting the amount of changes, and deciding what to send based on that number.

For large images (>1000 elements), using this method adds to the experience. A change feels near-instant instead of having to wait more than a second, disturbing the flow of thought.

Two methods of generating the delta have been discussed. ”Diff” uses somewhat more server time than the original method, but can make a huge difference on the client. This is the most simple method, and for the programmer nothing changes. ”Fused” is a lot more difficult to implement. The programmer barely notices the change, the only difference is that, instead of

\[(\text{AppliedTransformations } m) \ (\text{Maybe } (\text{Image } m)) \rightarrow (\text{Image } m, \text{ImageChanges } m)\]

The benefit of this method, is that the server does not need more computation time to generate the delta and the image than it needed to generate just the image. For this research, not everything of this method is implemented, so not every image is possible. When Graphics.Scalable is fully rewritten, it is likely that the efficiency will improve, since the server did not need more time generating the delta along with the image in the current version. It can then still be decided whether or not this delta is used.
6.1 Future work

When the delta is too big, it is more efficient to send the model of the image instead of a delta. This is not implemented yet, but necessary when it is being merged into the iTasks system. An optional adjustment that may result in an improved performance would be adjusting the delta generation algorithms such that they generate better deltas. If the first element is added or removed and served to the current algorithm, it might generate a whole branch full of changes, even though only one element changed, which is not desired. Furthermore, editing only the attributes of some element (which is supported by SVGChange) is not implemented, but can lead to better performance for some images. An other efficiency boost might be obtained by supporting moving elements to an other position.

Finally, if a low server time is crucial, "Fused" can be completed, such that it works for every image. This however requires rewriting a big part of Graphics.Scalable. When fully implemented, it might be somewhat slower than the version tested in this research, since some extra processing steps would be added.
References


cloud storage services. In ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 307–327. Springer.


7 Appendix

7.1 Benchmarks

7.1.1 2000 circles, incrementing 1

<table>
<thead>
<tr>
<th>Original version</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>105</td>
<td>4</td>
<td>1026</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>4</td>
<td>993</td>
</tr>
<tr>
<td>3</td>
<td>114</td>
<td>4</td>
<td>1026</td>
</tr>
<tr>
<td>4</td>
<td>121</td>
<td>4</td>
<td>1005</td>
</tr>
<tr>
<td>5</td>
<td>115</td>
<td>4</td>
<td>1009</td>
</tr>
<tr>
<td>6</td>
<td>125</td>
<td>4</td>
<td>1016</td>
</tr>
<tr>
<td>7</td>
<td>113</td>
<td>4</td>
<td>1076</td>
</tr>
<tr>
<td>8</td>
<td>111</td>
<td>4</td>
<td>1014</td>
</tr>
<tr>
<td>9</td>
<td>128</td>
<td>4</td>
<td>1031</td>
</tr>
<tr>
<td>10</td>
<td>131</td>
<td>4</td>
<td>1012</td>
</tr>
<tr>
<td>average</td>
<td>118</td>
<td>4</td>
<td>1021</td>
</tr>
<tr>
<td>Diff</td>
<td>Server (milliseconds)</td>
<td>Communication (characters)</td>
<td>Scripting/Rendering (milliseconds)</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>-----------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>197</td>
<td>408</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>153</td>
<td>409</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>170</td>
<td>409</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>409</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>167</td>
<td>409</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>181</td>
<td>409</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>171</td>
<td>409</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>164</td>
<td>409</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>184</td>
<td>409</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>166</td>
<td>409</td>
<td>8</td>
</tr>
<tr>
<td>average</td>
<td>170</td>
<td>409</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fused</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>105</td>
<td>408</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>114</td>
<td>409</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>135</td>
<td>409</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>113</td>
<td>409</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td>409</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>130</td>
<td>409</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>150</td>
<td>409</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>117</td>
<td>409</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>409</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>109</td>
<td>409</td>
<td>8</td>
</tr>
<tr>
<td>average</td>
<td>118</td>
<td>409</td>
<td>9</td>
</tr>
</tbody>
</table>
### 7.1.2 1000 circles, incrementing 100

<table>
<thead>
<tr>
<th>Original version</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>136</td>
<td>4</td>
<td>515</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>4</td>
<td>451</td>
</tr>
<tr>
<td>3</td>
<td>108</td>
<td>4</td>
<td>522</td>
</tr>
<tr>
<td>4</td>
<td>137</td>
<td>4</td>
<td>436</td>
</tr>
<tr>
<td>5</td>
<td>122</td>
<td>4</td>
<td>470</td>
</tr>
<tr>
<td>6</td>
<td>115</td>
<td>4</td>
<td>492</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
<td>4</td>
<td>545</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>4</td>
<td>529</td>
</tr>
<tr>
<td>9</td>
<td>106</td>
<td>4</td>
<td>560</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>4</td>
<td>601</td>
</tr>
<tr>
<td>average</td>
<td>124</td>
<td>4</td>
<td>512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diff</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>163</td>
<td>39504</td>
<td>206</td>
</tr>
<tr>
<td>2</td>
<td>186</td>
<td>39504</td>
<td>192</td>
</tr>
<tr>
<td>3</td>
<td>171</td>
<td>39504</td>
<td>177</td>
</tr>
<tr>
<td>4</td>
<td>191</td>
<td>39504</td>
<td>175</td>
</tr>
<tr>
<td>5</td>
<td>204</td>
<td>39504</td>
<td>178</td>
</tr>
<tr>
<td>6</td>
<td>205</td>
<td>39504</td>
<td>180</td>
</tr>
<tr>
<td>7</td>
<td>189</td>
<td>39504</td>
<td>153</td>
</tr>
<tr>
<td>8</td>
<td>182</td>
<td>39504</td>
<td>174</td>
</tr>
<tr>
<td>9</td>
<td>216</td>
<td>39504</td>
<td>162</td>
</tr>
<tr>
<td>10</td>
<td>222</td>
<td>39504</td>
<td>160</td>
</tr>
<tr>
<td>average</td>
<td>193</td>
<td>39504</td>
<td>176</td>
</tr>
</tbody>
</table>
### 7.1.3 Moving along a grid

<table>
<thead>
<tr>
<th>Original version</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>105</td>
<td>3</td>
<td>394</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>3</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td>3</td>
<td>377</td>
</tr>
<tr>
<td>4</td>
<td>125</td>
<td>3</td>
<td>391</td>
</tr>
<tr>
<td>5</td>
<td>148</td>
<td>3</td>
<td>329</td>
</tr>
<tr>
<td>6</td>
<td>134</td>
<td>3</td>
<td>355</td>
</tr>
<tr>
<td>7</td>
<td>133</td>
<td>3</td>
<td>292</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>3</td>
<td>319</td>
</tr>
<tr>
<td>9</td>
<td>106</td>
<td>3</td>
<td>318</td>
</tr>
<tr>
<td>10</td>
<td>97</td>
<td>3</td>
<td>332</td>
</tr>
<tr>
<td>average</td>
<td>120</td>
<td>3</td>
<td>351</td>
</tr>
<tr>
<td>Diff</td>
<td>Server (milliseconds)</td>
<td>Communication (characters)</td>
<td>Scripting/Rendering (milliseconds)</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>-----------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>161</td>
<td>774</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>173</td>
<td>775</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>144</td>
<td>775</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>143</td>
<td>775</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>168</td>
<td>775</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>161</td>
<td>777</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>145</td>
<td>777</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>152</td>
<td>777</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>172</td>
<td>777</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>163</td>
<td>777</td>
<td>13</td>
</tr>
<tr>
<td>average</td>
<td>158</td>
<td>776</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fused</th>
<th>Server (milliseconds)</th>
<th>Communication (characters)</th>
<th>Scripting/Rendering (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>133</td>
<td>774</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>117</td>
<td>775</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>122</td>
<td>775</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>124</td>
<td>775</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>775</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>137</td>
<td>777</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>103</td>
<td>777</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>127</td>
<td>777</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>109</td>
<td>777</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>119</td>
<td>777</td>
<td>13</td>
</tr>
<tr>
<td>average</td>
<td>119</td>
<td>776</td>
<td>17</td>
</tr>
</tbody>
</table>