Progressive imagery with scalable vector graphics

Georg Fuchs[a], Heidrun Schumann[a], and René Rosenbaum[b]

[a]University of Rostock, Institute for Computer Science, 18051 Rostock, Germany;
[b]UC Davis, Institute of Data Analysis & Visualization, Davis, CA 95616 U.S.A.

ABSTRACT

Vector graphics can be scaled without loss of quality, making them suitable for mobile image communication where a given graphics must be typically represented in high quality for a wide range of screen resolutions. One problem is that file size increases rapidly as content becomes more detailed, which can reduce response times and efficiency in mobile settings. Analog issues for large raster imagery have been overcome using progressive refinement schemes. Similar ideas have already been applied to vector graphics, but an implementation that is compliant to a major and widely adopted standard is still missing. In this publication we show how to provide progressive refinement schemes based on the extendable Scalable Vector Graphics (SVG) standard. We propose two strategies: decomposition of the original SVG and incremental transmission using (1) several linked files and (2) element-wise streaming of a single file. The publication discusses how both strategies are employed in mobile image communication scenarios where the user can interactively define RoIs for prioritized image communication, and reports initial results we obtained from a prototypically implemented client/server setup.

Keywords: Progression, Progressive refinement, Scalable Vector Graphics, SVG, Mobile image communication

1. INTRODUCTION

Vector graphics use graphic primitives such as points, lines, curves, and polygons to represent image contents. As those primitives are defined by means of geometric coordinates that are independent of actual pixel resolutions, vector graphics can be scaled without loss of quality. This presents a strong advantage compared to raster imagery and makes vector graphics very suitable for mobile image communication scenarios where a given graphics must typically be represented in high quality for a broad range of viewing devices.

One problem with vector graphics is that file sizes increase rapidly as content becomes more detailed, due to the fact that more primitives are required. This can degrade response times and efficiency in mobile settings due to increasing data complexity and transmission requirements. Similar challenges arise for raster images and have been addressed using progressive refinement schemes. Progressive refinement facilitates low-fidelity previews, minimizes redundancy and bandwidth requirements during transfer, and also affords semantic aspects like predefined or interactive Regions-of-Interest (RoI) and guided tours through the data.[1] Although the application of progression to vector graphics has already been proposed in cartography, taking advantage of the approach using widely accepted standards is still an open research question.

In this paper we show how to provide progressive refinement schemes based on the extendable XML-based Scalable Vector Graphics (SVG) standard format.[2] The proposed methodology is compliant to the standard. We introduce two methods that vary in their advantages and properties and support a broad range of functionality that is beneficial for interactive and demand-driven image communication, as Regions-of-Interest and Levels-of-Detail. We illustrate and justify the introduced technology with results we obtained from first experiments and conclude that the introduced technology bridges the gap between flexible and efficient raster and compliant high quality vector image communication.

We will review work related to progressive raster and vector imagery in the Section 2 in order to lay the foundation for our methods in Section 3. Details to the implementation and initial findings are presented in Section 4. Conclusions and objectives for future work close our contribution in Section 5.

Further author information (contact either author):
G.F.: E-mail: georg.fuchs@uni-rostock.de, Telephone: +49 381 498 7484
R.R.: E-mail: rrosenbaum@ucdavis.edu, Telephone: ++1 530 754 9470

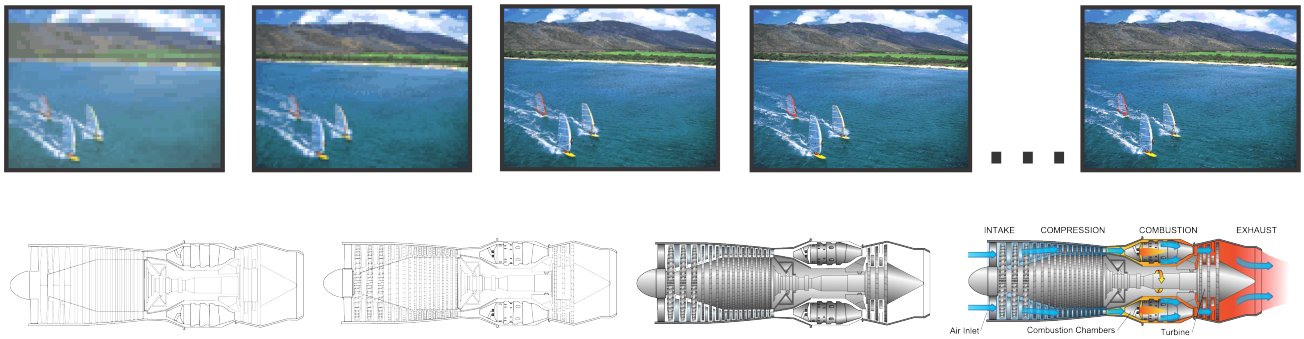

Figure 1. Progressive refinement (progression) provides first impressions of the content and may also be used to reduce consumed system resources. Although well-researched for raster imagery (top) applying progression to vector data (bottom) is still in its infancy.

2. RELATED WORK

Raster image communication has always been the main application of *progressive refinement* (*progression*). First schemes[3] mostly developed to overcome bandwidth constraints already allowed for content previews, but required redundant transfer and handling of image contents. This has later been overcome by the introduction of *scalable image compression and transmission* schemes. The basic idea here is to organize the encoded data-stream in such a way that the decoding of a truncated stream leads to a restored image with less detail.[4] Thus, content previews can be presented during a still running transmission. As all encodings of preview images are an integral part of the data stream created for the final image, a non-redundant data transmission is inherent. To combine the requirements of this procedure with a high compression performance, sophisticated codecs such as SPIHT[5] or JPEG2000[6] have been developed.

With progression as its foundation, image communication has greatly improved by the development of processing and transmission schemes supporting *Regions-of-Interest* (RoI) and *Levels-of-Detail* (LoD).[7] Not violating the general requirements of scalable compression, most important information is placed at the beginning of the stream followed by less important data. This leads to an *interest-ordered data-stream* and allows to decode and view important image regions earlier than others. Any valid stream termination leads to an image where important regions are presented exclusively or at higher quality than others. Dependent on the option to interactively define or parameterize their behavior during streaming, there are two main RoI types: static[6] and dynamic RoIs.[8] With a static RoI the streaming order is predetermined at compression time, while a dynamic RoI allows re-ordering (usually within limits) of the data stream upon decompression. Due to their ability to adapt to changing demands and the implemented paradigm - *Compress Once: Decompress Many Ways*[9] - dynamic RoIs are of exceptional importance in interactive environments.

Besides its more technical related objectives, progression has also been applied to support semantical aspects of image communication. It has been shown[1] that a well-designed, RoI-based refinement strategy can significantly improve the conveyance of selected image contents. Thus, progression can also be seen as a novel kind of information display combining a *predefined or interactive animation* of data contents with a highly resource-saving processing and transmission system.

Aside from overcoming bandwidth limitations for raster imagery, it has also been applied to other kinds of system constraints[10] and data.[11] Its application to geometric data in particular has been researched for a considerable time as well. Examples include approaches of Hoppe[12] and Lee et al.[13] providing means to simplify, respectively, large triangle meshes and complex iso-surfaces. Here, the resulting simplification hierarchy allows to reconstruct the object in various LoD to meet rendering hardware and/or fidelity requirements. However aspects of transmission, especially the use of common formats, were secondary to all of these approaches.

One application domain that has stimulated research in progressive vector transmission schemes are web-based GIS applications.[14] Here too, initial development focused on the delivery of spatial data as raster images, which could be reduced by the use of established image progression schemes, e.g., the one proposed by Rauschenbach and Schumann.[7] However, the desire to have access to more functionality, specifically direct querying and

manipulation of geographic objects, soon stimulated exploration of communication protocols based on vector formats.

The need to provide progressive refinement of vector maps to address bandwidth and device limitations has been recognized early. Bertolotto and Egenhofer[15] proposed a general framework for progressive vector transmission based on cartographic generalization, but did not consider encoding into a specific data format such as SVG. Similarly, Yang et al.[14] developed a hierarchical model for vector map data that facilitated progressive streaming of vertex data with client-side geometry reconstruction. Another frequent approach is to generate adapted maps at server-side according to request type and/or client profile;[16] "progression" here being understood as the transmission of maps with increasing details as the user selects successively smaller map sections, which however does incur redundancy in transmission. An example for this method is the scheme introduced by Costa et al.,[17] which uses SVG to encode adapted maps for transmission.

However, all these approaches discuss challenges and concepts primarily from the perspective of cartographic generalization, i.e., preservation of topological consistency and the complexity of real-time generalization. Progression is either based on proprietary data formats, or utilizes SVG merely to encode successive but self-contained maps for transmission.

Only recently, research efforts have been directed at obtaining progression schemes for SVG that would allow incremental refinement without incurring redundant transmission. These approaches exploit the fact that SVG provides means to refer to external resources, thus allowing to *fragment* a vector graphics across several files.

Li and Deng[18] utilize this for a multi-resolution mechanism for SVG-encoded maps based on a hierarchy of "SVG collections" each comprised of a base and one or more detail layers. Client-side scripting is employed to insert references to detail layers on request, which automatically triggers file streaming and updating visuals. However, their approach is inherently bound to the thematic layers provided by the backing GIS server. It does also not support dynamic RoIs to control what detail is shown where, which we consider a crucial property in mobile image communication.

In summary, progressive image communication based on raster data is well researched, but work towards progression schemes for vector-based graphics so far has mainly focussed on purely technical aspects neglecting compliant implementations. This is a limiting factor in communication scenarios that by principle require an agreement on protocol and data formats.

3. PROGRESSION SCHEMES FOR SCALABLE VECTOR GRAPHICS

The aim of our approach is to make all benefits that progressive data communication can provide – previews, efficient data compression and transmission, Tours-through-the-data, and support for interactive systems – available for arbitrary graphical content, as technical illustrations or visualizations, in vector format. In particular, we want to provide these features in a compliant manner for SVG as a data and transmission format with broad acceptance.

Conceptually, a dynamic progression scheme comprises the following steps (for a more detailed discussion, we refer to Rosenbaum and Schumann[19]): (i) creation and encoding of a (LoD) hierarchy on geometry at server side; (ii) flexible server-side sequencing of the hierarchy to facilitate dynamic (interactive) RoI functionality and (iii) transfer (streaming) to the client; (iv) identification of the individual chunks to allow (v) decoding and partial reconstruction of the image at client side. Figure 2 illustrates this general process.

With our approach, the SVG format is used to represent the hierarchy of the geometry required to apply progression to graphical contents. Following the general principle of scalable image coding, our objective is to re-order and stream this hierarchy in a way that decoding of a truncated stream leads to partially reconstructed content. In particular, the reconstruction of a truncated, re-ordered data stream is a fully compliant SVG image that can be processed and viewed by any SVG-capable tool.

In correspondence to the steps outlined above, therefore, two requirements exist at server-side: (1) the SVG must be structured so that its primitive hierarchy allows the extraction of different LoD and (2) organized in a way that permits transfer of primitive data for arbitrary RoIs. On client side it must be ensured that (3) received

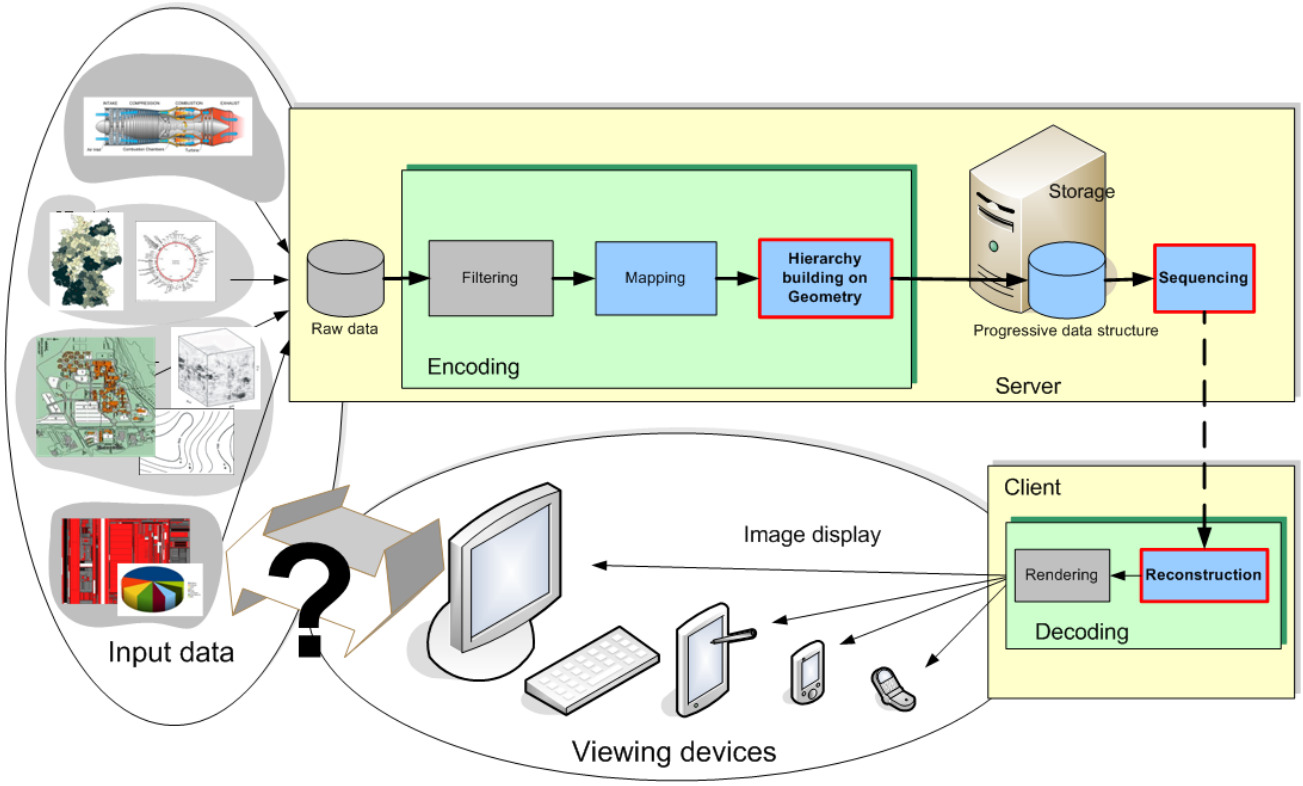

Figure 2. A data presentation scheme using progressive refinement for the scalable display of vector data (on the basis of Rosenbaum and Schumann[19]).

SVG fragments can be identified and re-integrated to form a single, standard-conformal file that describes the graphical content transferred up to that point.

After describing the *fundamentals of the SVG imaging standard*, we introduce two strategies that can achieve these objectives: (i) *progression using linked files* based on a decomposition of the original SVG into several linked files contributing to different levels of detail, and (ii) *element-wise progression* by dynamically decomposing the input file into its individual XML elements, whereas primitives are sorted according to semantic criteria. Due to constraints of the standard, each of the two strategies has its own advantages and drawbacks – and thus, application domains – in terms of transmission efficiency and client-side complexity. We provide a more detailed discussion of both strategies and its respective application domain in the following.

3.1 Fundamentals of the SVG imaging standard

Scalable Vector Graphics (SVG) is a modularized language for describing two-dimensional graphics in XML, endorsed by the WWW consortium.[2] SVG uses three types of graphic objects or *primitives*: geometric shapes (e.g., rectangles, circles, or paths), raster images, and text. Graphic primitives can be transformed, grouped, and styled by assigning presentation attributes like fill and stroke colors. Groups and individual objects are organized in a hierarchy controlling how graphical elements are composed during rendering. SVG has a well-defined rendering model, which is mandatory for compliant user agents (UA). SVG drawings can be interactive and dynamic. Animations can be defined and triggered either declaratively, i.e., by embedding SVG animation elements in SVG content, or via scripting. A rich set of event handlers can be assigned to any SVG graphic primitive. A supplemental scripting language further allows to modify the SVG Document Object Model (DOM), which provides complete access to all elements, attributes and properties. This facilitates sophisticated applications based on SVG.

SVG has been explicitly designed to offer broad interoperability with other web technologies and standards. For example, Cascading Stylesheets (CSS) can be used for styling. In particular, it offers extensive support

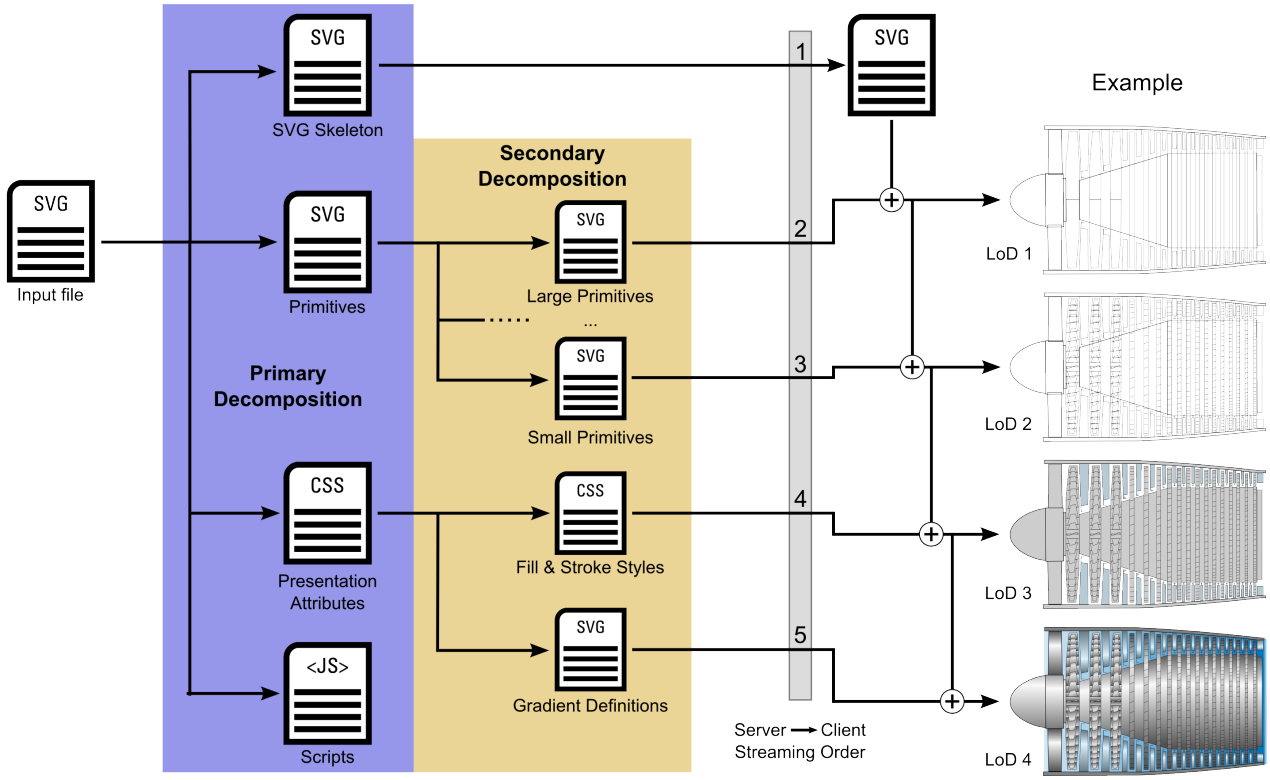

Figure 3. Progression scheme based on linked resource files (SVG fragments). The number of LoD corresponds to the fragment count. The client's SVG UA resource resolving mechanism automatically re-integrates fragments into the images as they are received in progression order. The right column shows an example of a jet engine compressor divided into its skeleton and four LoD fragments.

for integration with domain-specific languages (DSL) through XML namespaces[20] and linked resources, like stylesheets or embedded images, based on the XML linking language[21] (XLink).

To enhance scalability of SVG in the face of linked resources, the standard recommends that user agents optionally support what is called "progressive rendering".[2] This, however, is much more limited than the progression schemes proposed here in that it only addresses asynchronism of client-side DOM processing: the rendering process is either allowed to proceed by skipping unresolved nodes, i.e., with references only partially loaded or in error, or halted temporarily. The draft stipulates that rendering should be automatically updated or resumed, respectively, as soon as any incomplete node becomes available. While this automatism eases the adoption of more sophisticated progression schemes by accepting incomplete SVG files and managing rendering updates, in and of itself this mechanism does not account for the superordinate requirements of image communication as outlined in Section 2. In particular, SVG's progressive rendering has no notion of LoD, RoI, or re-ordering of data streams.

3.2 Progression using linked files

The first strategy exploits SVG's ability to incorporate linked external resources. Splitting up content over several linked SVG files is also referred to as SVG fragmentation.[18] For the purposes of progressive transmission of SVG files, the object hierarchy of an input SVG file is re-organized to represent the required LoD hierarchy as *content fragments*. Fragmentation involves a two-stage process performed once per input file in a pre-process (cf. Fig. 3). As the primary decomposition we introduce four higher-order fragment types to reflect the principle components inherent to SVG files: structure, geometry, styles and scripting/animation. The secondary decomposition further subdivides geometry and styling attributes into LoD.

The first fragment comprises all information required to describe the structure of a minimal valid SVG file. At the very least, this includes the XML header, SVG meta-data elements, the *<svg>* element presenting the object hierarchy root and the root element of the section for file-global definitions (see[2] for details). For non-empty images, the *<svg>* further encloses the hierarchy of object groups that contain links to the externalized content fragments. We refer to this base file as the SVG *skeleton*.

Individual graphic primitives (geometric shapes, raster images and text) and object groups – i.e., objects composed of multiple primitives, subgroups and transformations – make up the second type of fragments defining the geometric Levels-of-Detail. They are externalized into self-contained SVG files and replaced by resource references (*<use>* elements containing an XLink pointer) in the SVG skeleton file. By breaking up the object hierarchy into several individual fragment files according to application requirements, fine-grained control over the LoD available for progression can be achieved. Fig. 3 illustrates this for a technical illustration where two geometric LoD have been defined: shapes defining the outlines of major components were grouped into the fragment constituting the first LoD; smaller primitives provide structural detail. Other applications may require a breakdown into even more fragments for a corresponding number of geometry LoD.

Presentation attributes such as line strokes or colors gradients constitute the third fragment type. These styling information constitute the higher levels of detail in that they refine the outline view provided by the geometric primitives (cf. right column in Fig. 3). Presentation styles are often shared between multiple graphic primitives; also there usually is no expedient sorting order that would define a LoD hierarchy for them. For this reason, we externalize styles into just two fragment files based on the encoding format: one stylesheet containing all line stroke, solid fill color and text style definitions in CSS notation, and one SVG file containing all gradient and fill pattern definitions specified in SVG syntax. This may entail conversion of SVG's internal presentation attributes, i.e., XML element attributes,[2] into CSS notation.

The last fragment type comprises all scripts that enhance the SVG image with interaction functionality (e.g., in reaction to mouse input) and dynamic animations. Because they can modify almost any aspect of the SVG DOM, scripts are loaded only after all graphic primitives and presentation attributes have been transmitted to the client in previous progression steps. Script externalization is trivial; inlined script code can be copied to an external file and replaced by an XLink pointer in the corresponding *<script>* in the SVG skeleton.

After preprocessing the input SVG file into fragments, the server can serve this content progressively to clients by streaming the fragment files in order. The first file transmitted is always the SVG skeleton, followed by the graphic primitive fragments, then presentation attributes, and scripts last. The determination of the progression order in which fragments of one type should be transmitted to clients is part of the preprocess, in the same way that an appropriate decomposition of content into fragments is application-specific. This information thus is not intrinsic to the fragmented SVG content and therefore not stored in the skeleton. Rather, it is maintained server-side in a meta-data descriptor.

The client's SVG user agent handles necessary rendering updates automatically as fragments are received and decoded, i.e., become *resolved*, at client-side. This scheme therefore is compliant with the SVG standard.

In principle, dynamic RoI support is possible with this approach at the granularity of fragments: all fragments that intersect the current RoI are prioritized for transmission. However, this would require splitting graphic primitives into a relatively large number of spatially separated fragments, reducing efficiency, since each fragment requires its own SVG scaffolding. More importantly, this requires a means to signal RoI/LoD requests to the server, which is not provided by standard SVG UAs.

The advantage of the fragmentation-based progression strategy is that it is completely standard conformal. All fragments are valid SVG files; the progressive refinement is controlled exclusively server-side by the order in which fragment files are streamed, so an arbitrary SVG UA can be used at client side. In addition, because all fragments are transmitted as self-contained files, gzip compression (according to RFC1952, mandated to be understood by all SVG UA) can be employed to further reduce bandwidth requirements.

On the other hand, pre-processing of input SVG is required to convert object groups into externally referenced fragments. This is a partially manual task because what object groups constitute appropriate LoD is application-dependent. Number and size of fragments is a trade-off between LoD granularity and efficiency because of

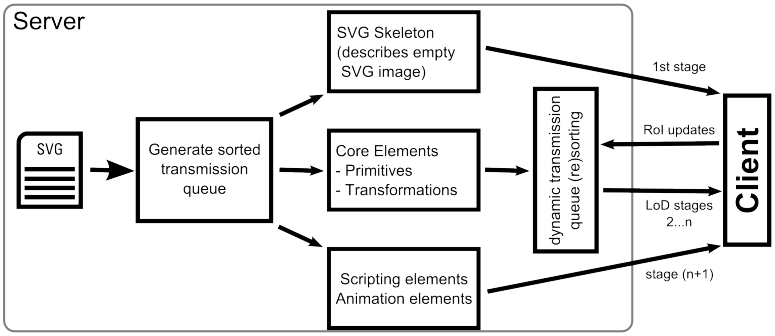

Figure 4. Per-element progression: the server scans the input SVG file on-demand to generate the SVG skeleton and a transmission queue of elements. Queue entries are successively transmitted to the client which re-integrates them into the skeleton. The client also signals changes to the RoI, which trigger a dynamic re-sorting of the remaining elements in the transmission queue.

increased SVG overhead since each fragment includes basic SVG scaffolding. This trade-off affects flexibility in progression order and RoI support, both of which scale with a larger number of smaller fragments.

Therefore, the fragment-based progression scheme is best suited for application scenarios where dynamic RoI support and fine-grained adjustment of progression orders is secondary to the need for supporting arbitrary SVG-capable clients without requiring specialized software components. Examples for these type of applications are communication scenarios where the refinement is defined in advance. This includes authoring-based content presentations as used in narrative illustrations. In this case refinement is mostly static providing limited means for interaction.

3.3 Element-wise progression

The second strategy addresses the limited flexibility in progression order and RoI support inherent in the fragmentation-based approach discussed above.

To transfer the SVG content into the LoD hierarchy, elements are decomposed into three higher-order detail levels: the SVG skeleton, graphic primitives, and scripting similar to the first strategy. This is illustrated in Figure 4.

The SVG skeleton contains almost the same information as described in Section 3.2, with one difference: the *<svg>* root element of the object hierarchy is always empty in the skeleton (cf. Fig. 5, left) as the entire object hierarchy is build up at client-side. The skeleton is always the first LoD transmitted to a client.

The set of graphic primitives constitute the second higher-order detail level. This level is further broken down into multiple second-order LoD to allow for progression. With the file-based approach, a fixed number of LoDs (by fragments) is generated during preprocessing. Contrary to this, in the element-based approach the number of detail levels corresponds to the the number of graphic primitives in the SVG file: each LoD is derived from the previous one by the addition of exactly one graphic primitive. No explicit pre-processing of the input SVG is required for this as the server dynamically traverses the SVG DOM to generate a sorted *transmission queue* of graphical primitives (cf. Fig. 4) depending on the current RoI and progression order.

The third higher-order level comprises scripts, animation elements and interaction event handlers. Because these may modify almost all elements and attributes of the SVG DOM, again, they are always inserted at the tail of the transmission queue.

During progressive refinement, the server transmits the skeleton first, followed by all SVG objects on the transmission queue according to the current sorting order. This sorting order depends on the current RoI. In doing so, elements are transmitted using minimal update packets, i.e., **not** as complete SVG files. Moreover, the server maintains a list of already transmitted DOM nodes, a "sent list", to prevent redundant transmission of data after the RoI has changed.

Figure 5. A synthetic example for an SVG skeleton and the update packet sequence transmitted by the server to progressively restore the input SVG file on the client. Note that only graphic primitives (numbered) are sorted. Intermediate hierarchy elements (boxed) are recursively injected into the queue as required to insert the respective child primitives.

Queuing update packets for transmission involves a two-step sorting process: first, graphic primitives are divided according to whether they intersect with the current RoI. Within both the RoI and non-RoI sets, primitives are further ordered according to a sorting criterion. If the RoI definition changes, the transmission queue is re-sorted to reflect the new progression order.

The definition of the sorting criterion is application-dependent. To this end primitives can be attributed with suitable attributes utilizing SVG's ability to integrate additional meta-data via domain-specific languages. If domain-specific ordering information is not available, we propose sorting according primitives' relative bounding box sizes instead, which results in progressive refinement very similar to that known from raster graphics. The reason for this is that primitives with a smaller extend usually contribute local details compared to large primitives constituting major image features.

The transmission queue only holds references to graphic primitives and script/animation objects as these comprise the progressive LoD. However, the source SVG object hierarchy includes additional structural elements, such as nested group nodes and geometric transformations, above those objects. Therefore, missing parent nodes of the element in the current update packet are recursively injected into the queue before the updated element itself. This is applied until an ancestor node that has already been transmitted is reached. The sent list is updated to include all injected DOM nodes besides the updated element so that subsequent updates to children with a common parent do not trigger its repeated transmission. After an update tuple has been successfully transmitted, the corresponding object is removed from the transmission queue.

The client component fulfills two functions: signaling changes to the RoI in reaction to user interaction and to perform element re-integration into the SVG DOM tree as these are transmitted in the form of update packets by the server. This functionality is not directly supported by standard SVG UA, instead requiring dedicated program logic running on the client to transform the stream back into a compliant representation. This can, however, be realized using SVG scripting exclusively as this allows for extensive modifications to the SVG DOM including insertion of new nodes. Section 4 describes a sophisticated AJAX-based implementation where the SVG skeleton is used as the transport container for the required client logic.

An update packet is a 3-tuple $\{parent\_id, child\_index, element\_data\}$ that contains, respectively, the DOM element identifier* of this element's parent node in the DOM, its child position index in the parent node, and the actual element data, i.e., the XML tag name, attribute list, and element content. Thus, update tuples allow for an exact re-creation of the source SVG object hierarchy independent of progression order. Maintaining this information is important because a node's child order does carry semantics under the SVG rendering model: child nodes are subsequently composed on top of the output from previous child nodes in index order. See Figure 5 for an illustration of this process.

Note that it is not necessary to handle references to external resources specially with our element-based approach. An element containing an XLink pointer (to a SVG fragment within a *<use>* element, a stylesheet, or

*The ID attribute mandatory for every SVG element is used for this purpose.

script) is transmitted by an update packet in the same way as all other objects. This simply results in a request for the referenced resource by the client's user agent as soon as the link-containing element has been re-inserted into the SVG DOM (cf. Section 3.1).

The advantage of the per-element approach is that it provides very fine-grained progressive refinement and efficient transmission through incremental updates with minimal meta data overhead. Decomposition of the input SVG file into its skeleton and update elements can be performed on-demand, not requiring a pre-process. This also allows to dynamically re-order the transmission queue to accommodate for interactive changes in the RoI definition. Another beneficial side-effect of this is that it allows to transmit and display a low-fidelity "thumbnail" preview of the complete vector graphics, such as the mini-map shown in Fig. 6. Finally, partially reconstructed streams are compliant SVG files that can be processed by any SVG-capable UA.

The penalty for this flexibility is a higher client load; it requires dedicated client logic for handling of update packets in addition to the subsequent processing performed by the actual SVG UA. In particular, per-element updates entail frequent DOM updates that are especially taxing for some resource-optimized XML parsers used in mobile environments.

The element-based refinement strategy allows for a much more dynamic RoI definition making it an ideal solution for highly interactive environments. This advantage is of benefit for all content browsing applications applying frequent changes in the displayed image region and LoD. The imposed burden in terms of required computing power, however, excludes client devices that are strongly constrained in their resource capabilities from the application of this approach.

4. IMPLEMENTATION AND INITIAL RESULTS

We prototypically implemented the element-wise progressive refinement strategy to prove its feasibility. It requires both a server component and client logic. We chose an AJAX (Asynchronous JavaScript and XML) based solution that allows to use a JavaScript- and SVG-capable web browser as user agent, e.g., Mozilla Firefox.

The server component has been implemented in Java. It uses a XML DOM parser to extract the SVG skeleton and an in-memory representation of the SVG object hierarchy from the input file. Graphic primitives are further analyzed to extract their axis-aligned bounding box[†]. Their area is used as an approximate measure of the primitive's size, which we use as a simple heuristic to determine the progression order. Also, bounding boxes are used to build an axis-aligned bounding-box tree to facilitate RoI intersection tests.

The server's web interface comprises of Java Server Pages (JSP). Communication between server and clients uses XMLHttpRequest, which is no W3C standard yet supported by all prevalent web browsers.

The client logic is a JavaScript that handles user interaction and DOM re-assembly from the update packets. This code is transfered to a client as integral part of the SVG skeleton, thus eliminating the need for an explicit installation procedure. After loading the SVG skeleton, the browser automatically executes the client logic script, and also assumes the role of the SVG user agent performing the SVG rendering.

User interaction supported in the prototype include defining and removing the RoI and stopping/resuming the progressive refinement. Defining the RoI is accomplishes by a click-and-drag mouse gesture to specify a rectangular area. This will cause primitives within this area to be transmitted first (cf. Fig. 6, 2nd left). Changing the RoI causes the server to re-sort the transmission queue accordingly (Fig. 6, 3rd from left). Removing the RoI by right-clicking the mouse will cause the transmission queue to revert to the default ordering in decreasing bounding box size. Stopping and resuming are achieved by key presses. Stopping progressive refinement before all LoD have been reconstructed can be is useful on small displays where too many details would cause clutter rather than further improving the visual representation.

Also shown in Fig. 6 is an example how auxiliary information not part of the original SVG image can be inserted to aid navigation of only partially reconstructed content. Here, a low-resolution thumbnail of the

[†]So far, the analysis is limited to SVG's six basic shapes (rectangle, circle, ellipse, line, polyline, polygon) and linear paths, but no curved paths or text nodes. SVG further supports a large number of coordinate units, however we currently support only pixels (px).

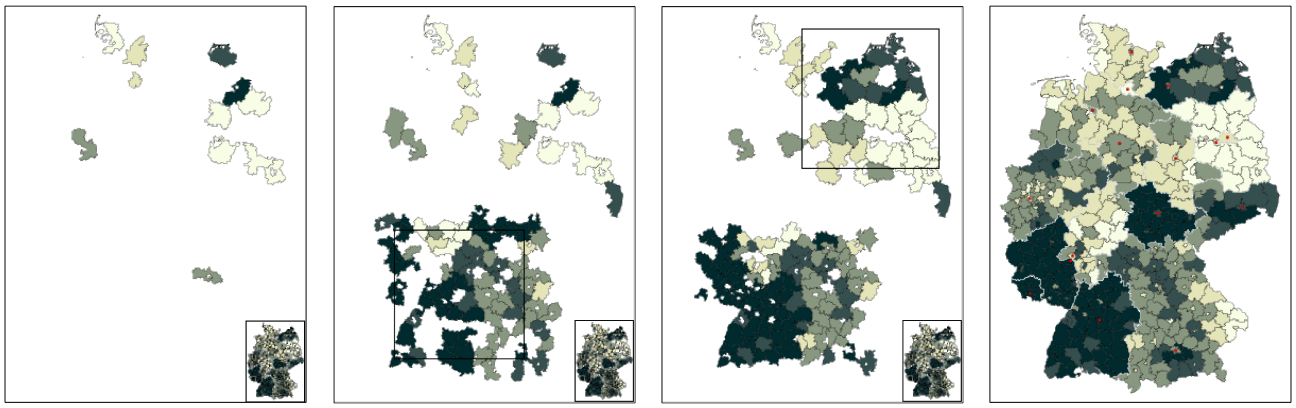

Figure 6. Example showing the element-wise progressive refinement of a map with primitives sorted by their relative bounding box sizes. Shown left is the reconstruction of the first few (smallest) primitives regardless of position. The middle two images show a dynamic change to the RoI (gray rectangles), prioritizing primitives intersecting the respective region. The mini-map in the lower right is not part of the input image but an auxiliary low-resolution raster image explicitly injected into the SVG DOM. (Map image from *wahlatlas.net*, used with permission under the Creative Commons Attribution licence CC-BY 3.0).

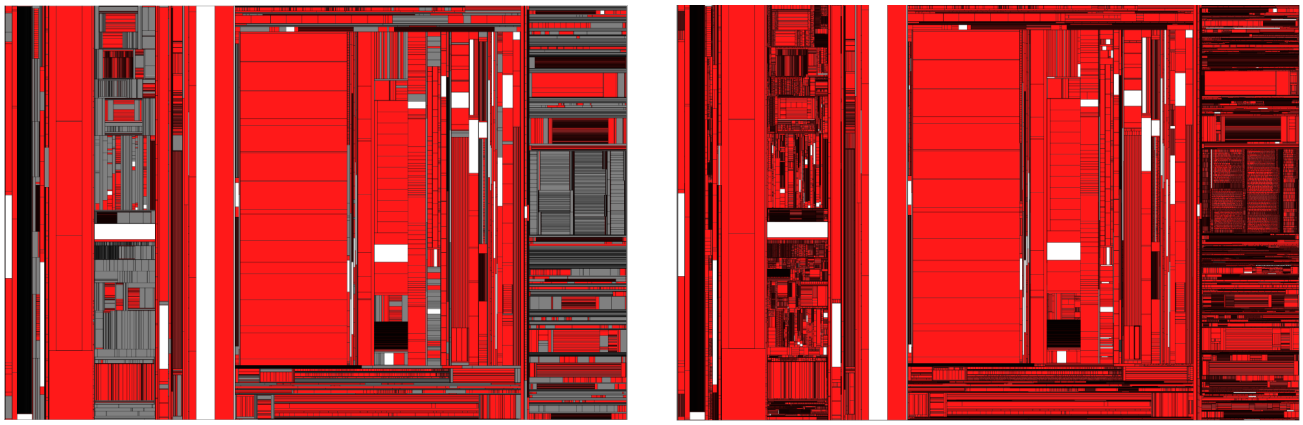

Figure 7. Example showing progressive refinement of a TreeMap visualization, using the element-wise progression approach. The left image contains 10 of 58 LoD (14.3% of the total transmission volume, cf. Table 1), right shows the fully reconstructed SVG for comparison.

complete map has been rendered into a raster image at server-side. This raster image is inserted into the SVG skeleton using a corresponding update packet to provide a "mini-map" style overview (Fig. 6, left). In this example, after the highest LoD has been transferred the overview is no longer needed, so it can be removed again. This is achieved by a special update packet that instructs the client to delete the DOM node with the corresponding node ID (Fig. 6, right).

Figure 7 shows another example for element-wise progression, this time for a TreeMap visualization[22] of a 7,600-node file system/directory hierarchy. We used this test case to roughly assess the efficiency of progressive streaming for large and complex SVG images. Using a hierarchical data set was a natural choice since it is straightforward to match the data with SVG's primitive hierarchy.

The simple SVG-TreeMap conversion we employed generated a total of 58 graphic primitives, i.e., it did not break the SVG hierarchy down to the data leaf level, but combined sub-trees beyond a certain level into a single multi-segment path object. Thus, this example image will yield 58 LoD using the element-wise progression scheme. Figure 7 also illustrates that even after only the first 10 LoD have been reconstructed, the image already presents a good overview on the data distribution. Note how missing data are indicated using a gray fill color by providing a default background fill color as part of the skeleton.

As can be seen in Table 1 the transmission of update packages incurs some overhead due to the parent id and

child index information. Note that the figures lists only the application data volume, without XMLHttpRequest protocol overhead.

Table 1. Comparison of relative data sizes (transfer volume) for the TreeMap visualization shown in Fig. 7. The unmodified input SVG file is listed for reference.

File	**Absolute size**	**Relative size**
SVG Skeleton (incl. client logic script)	40 kB	0.6%
Partially reconstructed content (10 LoD)	953 kB	14.3%
Fully reconstructed content (58 LoD)	6,963 kB	104.7%
Unmodified input SVG file	6,652 kB	100.0%

The fragmentation-based approach can be easily implemented as well. This requires a modified web server that handles client request for the fragmented SVG file. Here too, the client itself can be any HTTP- and SVG-capable UA. Instead of serving subsequent request for the fragments referenced in the SVG skeleton immediately and with equal priority, the modified server prioritizes fragment requests in the order mandated by the meta-data manifest (cf. Section 3.2). Streaming of lower-prioritized fragments is either suspended until all higher-priority fragments have been completely transmitted, i.e., strictly sequential streaming[‡]; or the highest-priority fragments are transmitted in parallel according to a specified bandwidth budget.

5. CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

Due to the constantly increasing data volumes, compliant and resource-efficient handling of image data becomes more and more important. We proposed two strategies that adopt progressive refinement as a foundation for an incremental handling and display of vector imagery for the widely applied standard SVG. *Progression based on linked files* requires low resources at client side, but offers little flexibility in the dynamic definition of RoIs. Thus it is well suited for low-power client devices and narrative illustrations. *Element-wise progression* is well suited for interactive applications, but requires additional computing power at client side. Both approaches allow for partially reconstructed images that are nonetheless compliant to the SVG standard. This greatly facilitates combination with other SVG-enabled technology for processing and display, as well as simple integration into existing systems. A proof-of-concept implementation shows these approaches are feasible with existing SVG user agents.

The proposed strategies are just the beginning of a wide variety of extensions that may be developed. Future development will be directed towards completing support for SVG graphics primitives (complex paths, alternate coordinate system units) as well as more efficient handling of the transmitted data by use compact binary encoding for update elements instead of strings containing verbatim SVG syntax. RoI support in the fragment-based approach can be improved by allowing interleaved transmission of geometry and styling fragments. This requires determining which styles are referred from graphics primitives within the RoI, i.e., a DOM analysis for automated "style localization". This would allow to attain full detail within the RoI before remaining geometry outside of the RoI has been transmitted.

Follow-up research will also focus on the application of the developed strategies to concrete application scenarios, in particular data visualization. Application of progressive SVG to visualization is promising for two reasons. First, large data often results in cluttered visualizations especially on small screens. Using vector representations with reduced geometric LoD are a promising solution here. Second, the ability of SVG to integrate other domain-specific languages facilitates annotation of visualization results with meta-data. This can be exploited in scenarios where visualization output is consumed, but not generated, on (small) mobile devices. Our aim is to develop an annotation concept that enables clients to re-parametrize, within limits, the visualization locally, i.e., without necessitating a re-transmission of the complete visual representation. The challenge here is to integrate meta-data annotations with the progressive geometry LoDs.

[‡]Note this applies to all fragments that are requested from the same server as the skeleton; absolute XLink pointers to remote resources can still cause requests to other servers in parallel.

REFERENCES

[1] Rosenbaum, R. and Schumann, H., "Progressive raster imagery beyond a means to overcome limited bandwidth," in [*Proceedings of Electronic Imaging - Multimedia on Mobile Devices 2009*], (Jan. 2009).

[2] Jackson (ed.), D., "Scalable Vector Graphics (SVG) 1.2 W3C Working Draft," (27 October 2004).

[3] Lohscheller, H., "A Subjectively Adapted Image Communication System," *IEEE Transactions on Communications* **32**, 1316–1322 (Dec. 1984).

[4] Lin, E., Podilchuk, C., Kalker, T., and Delp, E., "Streaming video and rate scalable compression: What are the challenges for watermarking," in [*Proceedings of SPIE Security and Watermarking of Multimedia Contents III*], **4314**, 116127 (2001).

[5] Said, A. and Pearlman, W. A., "An image multiresolution representation for lossless and lossy compression," in [*Transactions on Image Processing, vol. 5, pp. 1303-1310*], (1996).

[6] "JPEG 2000 image coding system, Part1," final publication draft, ISO/IEC JTC 1/SC 29/WG 1 N2678 (July 2002).

[7] Rauschenbach, U. and Schumann, H., "Demand-driven Image Transmission with Levels of Detail and Regions of Interest," *Computers and Graphics* **23**(6), 857–866 (1999).

[8] Taubman, D., "Remote browsing of JPEG2000 images," in [*Proc. IEEE ICIP2002*], I: 229–232 (Sept. 2002).

[9] Taubman, D. and Marcellin, M., [*JPEG2000: Image compression fundamentals, standards and practice*], Kluwer Academic Publishers, Boston (Nov. 2001).

[10] Pascucci, V., Laney, D. E., Frank, R. J., Scorzelli, G., Linsen, L., Hamann, B., and Gygi, F., "Real-Time Monitoring of Large Scientific Simulations," in [*Proceedings of ACM Symposium on Applied Computing (SAC 2003)*], (2003).

[11] Everitt, K. and Yee, K., "Phoebus: Progressive Display of Partial Query Results," in [*Research reports of EECS Computer Science Division, University of California, Berkeley*], (2002).

[12] Hoppe, H., "Progressive Meshes," *Computer Graphics* **30**(Annual Conference Series), 99?108 (1996).

[13] Lee, H., Desbrun, M., and Schrder, P., "Progressive encoding of complex isosurfaces," in [*SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*], 471476, ACM Press, New York, NY, USA (2003).

[14] Yang, B., Purves, R., and Weibel, R., "Implementation of progressive transmission algorithms for vector map data in web-based visualization," in [*International Archives of Photogrammetry and Remote Sensing*], Citeseer, Istanbul, Turkey (2004).

[15] Bertolotto, M. and Egenhofer, M. J., "Progressive vector transmission," in [*GIS '99: Proceedings of the 7th ACM international symposium on Advances in geographic information systems*], 152–157, ACM, New York, NY, USA (1999).

[16] Peng, Z.-R. and Zhang, C., "The roles of geography markup language (GML), scalable vector graphics (SVG), and Web feature service (WFS) specifications in the development of Internet geographic information systems (GIS)," *Journal of Geographical Systems* **6**(2), 95–116 (2004).

[17] Costa, D., de Paiva, A., Teixeira, M., de Souza Baptista, C., and da Silva, E., "A Progressive Transmission Scheme for Vector Maps in Low-Bandwidth Environments Based on Device Rendering," in [*Advances in Conceptual Modeling - Theory and Practice*], Roddick, J. e. a., ed., *Lecture Notes in Computer Science* **4231**, 150–159, Springer Berlin / Heidelberg (2006).

[18] Li, D. and Deng, L., "Multi-resolution Mechanism for SVG," in [*Asia-Pacific Conference on Information Processing (APCIP 2009)*], **2**, 139 –143 (July 2009).

[19] Rosenbaum, R. and Schumann, H., "Progressive refinement - more than a means to overcome limited bandwidth," in [*Proceedings of Electronic Imaging - Visualization and Data Analysis 2009*], (Jan. 2009).

[20] Bray, T., Hollander, D., Layman, A., and Tobin, R., "Namespaces in XML 1.0 (Second Edition)," (2006).

[21] DeRose, S., Maler, E., and Orchard, D., "XML Linking Language (XLink) Version 1.0," (2001).

[22] Shneiderman, B., "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Transactions on graphics (TOG)* **11**(1), 92–99 (1992).