Support for Arbitrary Regions in XSL-FO

A proposal for extending XSL-FO semantics and processing model

Ana Cristina B. da Silva
Joao B. S. de Oliveira
-
PUCRS/FACIN
{benso,oliveira}
@inf.pucrs.br

Fernando T. M. Mano
Thiago B. Silva
Leonardo L. Meirelles
PUCRS/CPSE
{fernando@cpts.pucrs.br}

Felipe R. Meneguzzi
Fabio Giannetti
-
Hewlett-Packard
{felipe.meneguzzi,
fabio.giannetti}
@hp.com

ABSTRACT

This paper proposes an extension of the XSL-FO standard which allows the specification of an unlimited number of arbitrarily shaped page regions. These extensions are built on top of XSL-FO 1.1 to enable flow content to be laid out into arbitrary shapes and allowing for page layouts currently available only to desktop publishing software. Such a proposal is expected to leverage XSL-FO towards usage as an enabling technology in the generation of content intended for personalized printing.

Categories and Subject Descriptors: I.7.2 [Document and Text Processing]: Desktop publishing, Format and notation, Markup languages, Photocomposition/typesetting

General Terms: Arbitrary Shapes, Typesetting, Digital Publishing

Keywords: XSL-FO, XML, LaTeX, SVG, Arbitrary Shapes

1. INTRODUCTION

The XSL-FO standard [13] describes XML documents separating content and layout information. This has led it to be considered an interesting alternative for publishing workflows [8]. One of the main advantages of using XSL-FO in publishing lies in it being an open standard based on XML. Moreover, the processing of a document based on this standard can be logically broken down into multiple stages that can be distributed among specialized service providers.

From a formatting point of view, the XSL-FO format provides constructs for specifying page layouts in which content flows can be positioned automatically. Such capability is important as it eases the process of paginating complex content. Nevertheless the current version of the standard has limitations regarding the type of layout over which flow content can be placed. In particular, XSL-FO version 1.0 [13] defines five possible regions within a page, only one of which can be used for the disposition of flow content. Such limitation is overcome in version[1] 1.1 [14] through a construct called **fo:flow−map**, which allows multiple content flows to be mapped into specific regions within a page.

Despite such improvements in version 1.1, the current page model allows only rectangular regions in which content is laid out. If XSL-FO is to be used as the base format for a Digital Printing workflow, more flexibility in the definition of content holding regions is required. Therefore, we propose to define regions using arbitrary shapes as a means to allow for more flexible content layout within XSL-FO.

This article describes an extension of XSL-FO 1.1 [14] that allows the definition of an unlimited number of page regions, each one having arbitrary geometric shape. Such an extension empowers the standard with document typesetting capabilities with complex graphical compositions.

In the related work section we provide a brief overview of the related work used in the conception of this proposal; Section 3 lays out the set of representational possibilities intended for the arbitrary regions and define the new formatting objects that will be used to that end; Section 4 describes the extensions required by the area tree model to accommodate the new rendering possibilities; in Section 5 we briefly describe an implementation of an XSL-FO rendering engine that supports the proposed extensions, and finally, Section 6 discusses the proposal and its implementation so as to provide the basis for its refinement and further development.

2. RELATED WORK

In the following sections we discuss a series of document description languages, with a special emphasis on those that decouple content and presentation details (Sections 2.1, 2.2, 2.3 and 2.4). Such languages are of particular interest in the context of Digital Publishing and the production of high quality personalized documents, as that separation allows a seamless integration between user specific content, document content templates and styling information.

2.1 XSL and Formatting Objects

Extensible Stylesheet Language (XSL) is a standard devel-

[1] A W3C working draft at the time of this writing.

oped and maintained by the World Wide Web Consortium (W3C) whose main purpose is to provide a means through which XML data can be formatted for presentation in multiple media types [13]. The standard itself is divided into two XML vocabularies: XSL itself, and the *XSL-Formatting Objects (XSL-FO)*. Each one of these vocabularies is associated with a distinct process during the conversion of XML data into a presentation in the output media.

The stylesheet language included in the standard is intended to provide a mechanism to modify an arbitrary XML tree into one described in terms of elements within the XSL-FO namespace. Such a process is performed by an XSL processor and is called *Tree Transformation*. The resulting XSL-FO tree represents non-paginated content with formatting instructions, which can be *formatted* into a specific presentation format and medium (*e.g.* PDF [3], PS [2], . . .), following the formatting semantics described in XSL-FO. The specification of a document using FO directives is composed of three top-level sections (Figure 1):

- An **fo:layout−master−set** element holding the geometric definitions of the FO regions used throughout the document as well as contexts in which these definitions are used;

- An **fo:declarations** element holding global declarations for the document, and;

- An **fo:page−sequence** element holding the specification of sequences of content that will be distributed in pages within a document. Content for these page sequences is specified as either **fo:static−content** for content that is not intended to be broken down into multiple pages or **fo:flow** elements for content that spans across multiple pages.

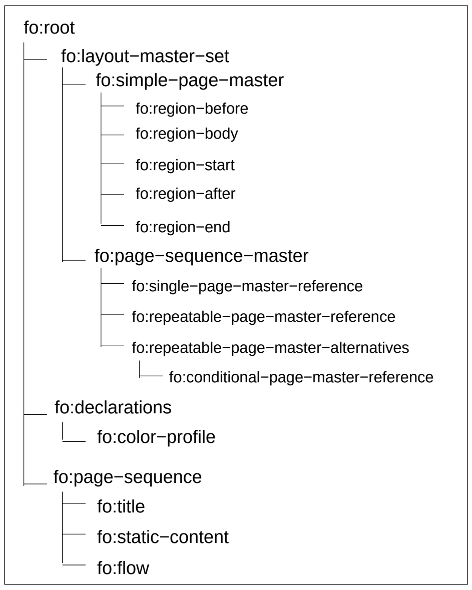

Figure 1: Sections of an XSL-FO Document.

2.2 TeX and LaTeX

TeX [7] (and its variants) is a typesetting system developed by Donald Knuth aiming at the production of high-quality documents for press printing. TeX document production is driven by a *document class* which specifies the general document structure as well as presentation details. Coupled with a content file, it generates a formatted version of such content according to the instructions of the document class. LaTeX is a document preparation system that builds over TeX typesetting language aiming to decouple formatting from content, facilitating the production of a large volume of consistent high-quality output.

With regards to document layout specification, TeX uses a set of imperative primitives to define rectangular areas in which content is laid out. Commands for specifying complex layouts are usually defined in terms of the low-level primitives included in the native TeX implementation and incorporated into a given document through macro packages, such as the one used in LaTeX.

Considering that the native layout model is based in rectangular areas, the inclusion of complete arbitrary shapes processing capabilities cannot be easily included using its original layout primitives. In particular, TeX low-level commands can be used in the creation of arbitrarily shaped paragraphs or even pages for the layout of a single flow of content, but this is not a trivial task and would push the TeX interpreter model to its limits. In this respect, a set of macros called `shapepar` [12] partially provides such a functionality enabling the layout of single-paragraphs within a given shape. Nevertheless, such macros do not allow flowing text to be laid out throughout multiple pages, or the specification of multiple arbitrary content regions within the same page. Moreover, LaTeX also includes a `parshape` command, which given a set of constraints to line indentation, size and number, can generate shaped paragraphs. Again, such a command requires the involved constraints to be calculated outside LaTeX in a non-trivial process.

2.3 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is a language for describing two-dimensional graphics in XML [16] developed jointly by the W3C, Adobe, Canon among other contributors and maintained by the W3C. Version 1.1 of this standard specifies primitives for drawing vector graphics, raster images and non-paginated text. SVG is loosely related to PostScript in its graphical primitives, as their specification is very similar to this language.

Even though the current SVG specification does not aim to provide a complete document description and printing language such as PostScript or PDF, the working draft of SVG 1.2 [15] shows a clear tendency towards turning SVG into a full-fledged printing vocabulary. Evidence of such direction is the inclusion of primitives for the specification of paginated content, as well as for flow content and line-breaking capabilities. Moreover, SVG 1.2 text-handling capabilities will be extended to allow the rendering of text paragraphs within arbitrary shapes. In its current form, however, SVG does not provide mechanisms for content-flow across multiple pages or multiple regions within the same viewport.

2.4 Other

This section briefly discusses other related languages used for document description, more specifically PostScript and PDF. These languages are used extensively as output formats for document generation, but their low-level graphical primitives prevents them to be useable as user-specified document languages.

2.4.1 *PostScript*

The PostScript language is a programming language designed to convey a description of virtually any desired page to a printer [1]. As a programming language it must be interpreted by an appropriate interpreter program which implements the semantics for its execution environment, which allows the use of variables and the combination of basic operators into complex functions and procedures. Moreover, it is essentially a low-level control language for usage directly at the printer level as an automatically generated format. Using the features of the language, it is nevertheless possible to describe shapes (using straight lines or parametric curves, for example) and provide a set of PostScript procedures to take such shapes and some text, filling the shape with the text and positioning the text at the proper places. However, solving the problem with this approach might be both expensive (as PostScript processing is not as fast as other languages) and complex (as one has to deal with numerical computations that are sensitive to small differences between seemingly similar shapes).

2.4.2 *Portable Document Format*

The Adobe Portable Document Format (PDF) is a file format for representing documents in a manner independent of the application software, hardware, and operating system used to create them and of the output device on which they are to be displayed or printed [3]. PDF documents are described in terms of a stream of objects, be they static printable content or other kind of electronic content or metadata. It is generally used as a container for other content specification formats such as PostScript, raster images, audio, or other interactive content. Notwithstanding its extended functionality over lower level formats, it is still a generally low-level description language employed mainly as automatically-generated output format, and suffers from the very same limitations as PostScript as far as arbitrary content layout representation is concerned.

3. ARBITRARILY-SHAPED FORMATTING OBJECTS

In this section wea describe an extension to the XSL-FO standard that allows the specification of any number of page regions within arbitrarily-shaped page regions. We therefore propose a conservative extension to XSL-FO 1.1 [14], *i.e.* a valid XSL-FO document is also a valid document within our extended semantics. Thus, the relevant aspects of XSL-FO 1.1 are described in Section 3.1. A modified page model is described in Section 3.2, whereas the elements comprising our extension and their associated attributes are described in Section 3.3. As we describe the new elements, examples will be provided and the associated attributes will be introduced as well as their semantics and association to the geometrical structure of the document.

3.1 XSL-FO Basics

In XSL-FO version 1.0 [13] the only page region in which flow content could be positioned is the **fo:region−body**, thus the content placed on all the other regions was limited to static headers, footers or similarly constant material. As a consequence, the number of pages of a given sequence was dictated by the amount of flow content included within the body region of such page sequence. Such an association of flow content only to the **fo:region−body** is expected to be overcome in version 1.1 of the XSL-FO standard [14]. In the current working draft the **fo:layout−master−set** element is augmented with a construct called **fo:flow−map**. meu construct describes a mapping between named page regions and named content flows. In order to maintain compatibility, a mapping representing the region to flow organization of XSL-FO 1.0 is assumed as default in case no flow map is supplied. A flow map example representing the default mapping is described in Figure 2.

```
<fo:flow−map flow−map−name="default−mapping">
 <fo:flow−assignment>
  <fo:flow−source−list>
   <fo:flow−name−specifier
    flow−name−reference="xsl−flow−start"/>
   <fo:flow−name−specifier
    flow−name−reference="xsl−flow−body"/>
  </fo:flow−source−list>
  <fo:flow−target−list>
   <fo:region−name−specifier
    region−name−reference="xsl−region−start"/>
   <fo:region−name−specifier
    region−name−reference="xsl−region−body"/>
  </fo:flow−target−list>
 </fo:flow−assignment>
</fo:flow−map>
```

Figure 2: Flow map within XSL-FO 1.1.

Using a flow map, it is possible to attach content flows to any one of the XSL-FO page regions. The possibility of mapping multiple content flows in a page sequence into multiple page regions represents the possibility of not being able to determine the definition of the total number of pages by any particular flow. Thus, depending on the amount of content within a flow and the size of its encasing page region a given flow may or may not command the number of pages generated by a given page sequence.

3.2 Page Model

The new page model for the proposed extensions allows the specification of any number of content-bearing page regions as well as the five regions specified in XSL-FO 1.1. These additional regions may possess an arbitrarily complex geometric specification, limited only by the language chosen for its description. Furthermore, our departure from the original XSL-FO page-region organization implies that overlapping of arbitrary portions of multiple adjoining regions is possible. An example of such layout is shown in Figure 3.

In order to cope with the new possibilities, the proposed representation allows regions to have depth values so as to allow region areas to be prioritized. A region can also have a specific behavior associated to it which is considered when

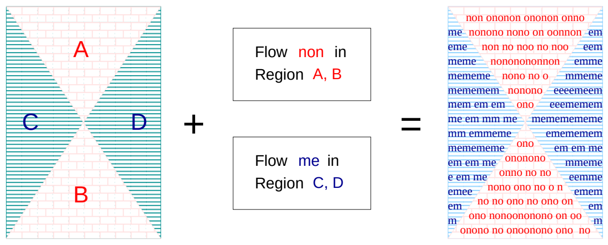

Figure 3: Possible page layout in extended XSL-FO.

overlapping with other regions occur, thus allowing a user to control the interaction among overlapping regions.

3.3 New Formatting Objects

The description of arbitrary page regions in the extended XSL-FO specification is accomplished adopting two mechanisms similar to the currently supported pagination model. As previously seen there are two ways of mapping a flow inside a page region, a direct one using the region-name as referencing attribute and an indirect one using the flow mapping. The proposed approach accomplishes a similar effect regarding the association of shapes to regions enabling the direct embedding of a shape description within a new region-arbitrary element as well as a region mapping.

3.3.1 Direct Mapping through **fo:region−arbitrary**

In the direct mapping approach, an **fo:region−arbitrary** element, declared as a child of the **fo:simple−page−master** element is introduced. Unlike its XSL-FO 1.1 siblings, any number of different **fo:region−arbitrary** elements can be declared within a given **fo:simple−page−master** as long as they are uniquely named. The geometric outline of an arbitrary region is specified using an external vector format. Such specification is included as a child element of the arbitrary region element, in case the format is XML-based or as CDATA otherwise (*e.g.* PostScript). Within our proposal the chosen format is SVG [16] due to its XML nature and consequent easier adaptation of an XSL-FO parser to cope with it. For example, an SVG-specified arbitrary region named Region1 in Figure 4.

```
<fo:region−arbitrary region−name="Region1" ...>
  <svg ...>
    ...
  </svg>
</fo:region−arbitrary>
```

Figure 4: Arbitrary Region SVG specification.

3.3.2 Indirect Mapping through **fo:shape−map**

The support for indirect mapping adds flexibility to the shape reuse across multiple pages and page sequences. Each shape will be defined inside an **fo:shape** element as part of the layout master set definition. Figure 5 shows the markup example.

This mapping can be applied over legacy regions to support non-rectangular shapes in the "border" layout, already

```
<fo:shape−map shape−map−name="document−regions">
   <fo:shape−assignment>
      <fo:shape−source shape−name−reference="poly1"/>
      <fo:region−target region−name−reference="left"/>
   </fo:shape−assignment>
   <fo:shape−assignment>
      <fo:shape−source shape−name−reference="poly2"/>
      <fo:region−target region−name−reference="body"/>
   </fo:shape−assignment>
</fo:shape−map>
```

Figure 5: Shape Map specification.

available in XSL-FO 1.1, or in combination with the **fo:region−arbitrary** to achieve a "free-form" layout.

3.3.3 Free-form layout versus Border Layout

In the original XSL-FO model only four regions can be described (roughly corresponding to the header, footer and both margins of a page) and all that is not included in one of these regions comprises the body of the page. A similar effect can be immediately achieved through the use of four shapes. Figure 6 shows three page models that could be obtained through the use of arbitrary shapes. The first model imitates the standard XSL-FO model, with only four rectangular regions and the text body being defined as everything that is not included in those regions. In the second page there are two non-rectangular margins and the remaining area forming the body of the page is shaped as a Z. In the third page a non-rectangular header frames two other shapes that can be used as text columns. In this case the true body of the page is the remaining area around the columns, and will not have any content.

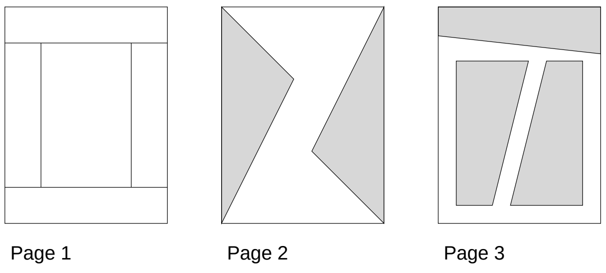

Figure 6: Simple arbitrary shapes.

For all practical purposes, it is important that the specified shape is composed solely of closed curves, that is, having inside and outside regions, possibly having self-intersections. Figure 8 shows two different shapes and their bounding boxes; the first shape is composed by a single primitive whereas the second one by three. Accordingly, the description of the first shape in Figure 8 in SVG would be given by the specification of Figure 7.

The representation of an unbounded number of page regions whose enclosing area is defined by any kind of closed curve represents a major departure from the original XSL-FO model of area overlapping semantics. More precisely, XSL-FO defines four border page regions, each of which specified in terms of an extent perpendicularly measured

```
<fo:region−arbitrary region−name="Object1" ...>
 <svg:svg width="11.7in" height="8.3in" viewBox="0 0 20157
       13858">
  <svg:g style="stroke−width:.025in; stroke:black;  fill:none "
        >
   <!−− Polygon −−>
   <svg:path d="M 3307,1653 2834,1181 2362,1181 2362,944
         1889,944 1889,1181 1417,1181 944,1653 1417,1653 1653
         ,2125 1417,2362 1417,2834 1653,2598 2598,2598 2834
         ,2834 2834,2362 2598,2125 2834,1653 3307,1653" />
  </svg:g>
 </svg:svg>
</fo:region−arbitrary>
```

Figure 7: Specification of Shape 1 in Figure 8.

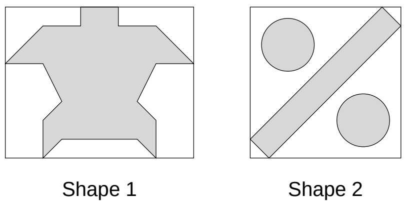

Figure 8: Two different shapes composed by a different number of primitives.

from the corresponding page border, as well as a body page region occupying the remainder of the space taken up by the border regions. As a result, the four border regions have clearly defined overlapping combinations, *i.e.* each border region can overlap one of its two adjacent borders (Figure 9).

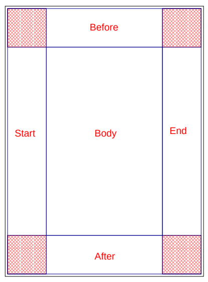

Figure 9: Overlapping regions in XSL-FO 1.0 denoted in dashed areas.

On the other hand, an unlimited number of possibly overlapping page regions entails that an unlimited number of region overlapping combinations may take place. Moreover, the possibly non-rectangular shape of the overlapping areas implies that when such an overlapping occurs, the clipping of areas must be performed through constructive area geometry instead of simply adjusting the dimensions of a rectangle, as is currently done in XSL-FO. We therefore propose two additional attributes for the arbitrary region element, which should enable the proper specification of overlapping behavior for non-rectangular areas. These attributes are described in Sections 3.3.4 and 3.3.5.

3.3.4 The layer *attribute*

In order to deal with the problem of resolving region-to-region interaction when overlapping occurs, we propose a layering approach specified through a mandatory layer attribute, and associated with the region element. The rules associated with layer are:

- The layer is a positive integer and provides the order for placing the region contents on a printed page. Content with the largest layer is placed first, the second largest layer is placed on top of the first, and so forth;
- Every region has a different integer associated to layer . If a number of disconnected shapes are required to be at the same layer on a page, they can all be placed within a single geometric description as in the second shape of Figure 8 above.

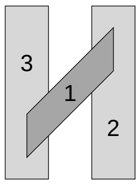

Figure 10: Three different shapes with overlapping regions.

Figure 10 contains an example composed of three regions placed on a page according to their layer. The number within each region represents the layer associated to it, but from that figure it is not clear how the three regions should interact in the overlapping areas. Figure 11 shows two different ways of resolving the same situation. The main difference is that the document designer could wish that overlapped regions should recede and render their content in the remaining visible area as in 11(a) or that their content is to be overlapped as in 11(b). A combination of interactions could also be desirable, with the right vertical bar receding and the left one being overlapped. To handle such situations, a second attribute is associated to a region, informing whether that region will recede or not.

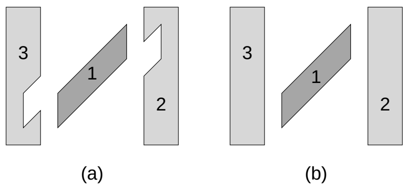

Figure 11: Two possible interactions between overlapping regions. In (a), the overlapped regions recede and text is not rendered in the overlapped areas. In (b) there is true overlapping.

The semantics of the layer attribute is very similar to that of the z−index attribute, which is already in the XSL-FO standard. Therefore, modifying the semantics of z−index and using it instead of layer is also an option.

3.3.5 *The* recede *attribute*

It is also necessary to provide a way of describing whether a region will recede in case some other region at a higher layer overlaps it. To store that information, a boolean recede attribute is also associated to a shape, with the following semantics:

- recede = false means that its associated region will keep its original geometry and not be affected by any other region on a layer above it. In this case, its content will be rendered in the original area.

- recede = true means that the current region will recede if overlapped by any other region on a layer above it. In this case, as much of its content as possible will be rendered in the remaining area, if any.

It is important to point out that the recede attribute only has meaning with respect to regions placed above the current region. As this attribute is not mandatory, an appropriate default value would be recede = false when such attribute is not declared.

3.3.6 *Other Attributes*

Beyond the additional attributes described in the previous sections, some of the attributes currently associated to region elements within a page master must have their semantics adapted in order to cope with the notion of arbitrary shapes introduced in this proposal. Therefore we also provide a set of semantic modifications for the existing attributes in order to cope with the introduced concepts.

Common Border, Padding and Background - When a background-image is defined for an arbitrarily-shaped region, the background-attachment property is always treated as if being defined as "fixed" as there is no defined method for scrolling arbitrarily-shaped areas. A tiled background will be positioned and distributed regarding the virtual rectangular area in which the arbitrary shape is inscribed, starting at the before-start corner of the area, and following the specified reference-orientation and writing-mode, and then subsequently clipped to the arbitrary shape. When vertical and/or horizontal position of a background is specified, it will always be calculated relative to the virtual rectangular area in which the arbitrary shape is inscribed, and the background will be subsequently clipped to the arbitrary area (Figure 12). Border color, style and width properties for specific edges (*e.g.* before, end, . . .) have no effect when specified for an arbitrary area. Instead, border-color/style/width properties are used to specify the same border behavior for all the edges of arbitrarily-shaped areas. Padding properties will have no effect for arbitrary areas, being always considered zero.

Layout-related Properties - The clip property is automatically set to auto for regions defined as arbitrary shapes with a recede property set to true. These attributes remain unchanged otherwise.

Pagination and Layout Properties - When the overflow property is defined for an arbitrarily-shaped region, the "scroll" value has no effect, as there is no defined semantics for scrolling such regions. The "visible" and "auto" values defer region clipping behavior to the recede and layer properties for arbitrary areas. The "hidden" value automatically sets the recede property to false for arbitrary areas in case such property has not been defined, otherwise it has no effect.

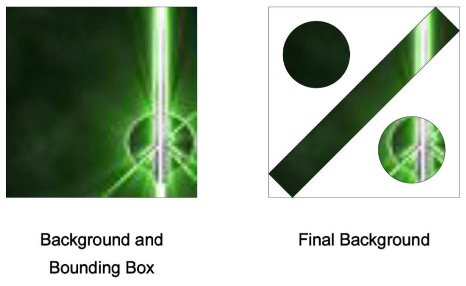

Figure 12: Background in Shape 2 of Figure 8.

3.4 Modified Layout Process

Considering the modifications introduced by our extended XSL-FO model, a number of aspects associated to the layout process described in the standard [13, 16] require adaptation. In particular, the process of generating the viewport/reference pairs for the arbitrary regions defined by the proposed extensions is modified into a generic model that also contemplates the standard page regions defined by the original standard [13]. Furthermore, the processing of arbitrary areas require different methods than the ones currently used to handle rectangular geometric constructs, in particular we chose to use primitives from Constructive Area Geometry (CAG) [10] to perform such handling.

3.4.1 *Region Interaction*

The introduction of arbitrary shapes in the description of page regions led to the region-to-region interaction model described in Section 3.3. Such interaction model requires a different process for the definition of the resulting areas, in particular one that handles the unboundedness of region overlapping situations while conforming to the layering and overlapping strategy specified by the user. We therefore propose Algorithm 1 as a reference for the handling of region-to-region interaction.

Algorithm 1 Region interaction resolution.

```
Let $R_i$ be a region with layer $= i$
Let $Sh_i$ be the shape associated to region $R_i$
Let $\setminus$ be the CAG difference operator
Let $n$ be total number of regions within a page master
for $i$ from $n$ to 0 do
  if recede $= true$ in $R_i$ then
    for $j$ from $i$-1 to 0 do
      $Sh_i = Sh_i \setminus Sh_j$
    end for
  end if
end for
```

3.4.2 *Conversion from standard regions*

Besides handling the interaction among arbitrary regions, the algorithm defined in Section 3.4.1 provides a generic solution within which the standard XSL-FO rectangular regions can be handled. Moreover, an implementation of such standard would also take advantage of a unified region model as it simplifies the mapping of regions in the FO tree into elements within the area tree. We therefore define a mapping from the five regions present in XSL-FO 1.0 into the proposed arbitrary model.

Definition 1 (XSL-FO Page master). *We define $\mathcal{P} = \langle \mathcal{P}_h, \mathcal{P}_w, \mathcal{P}_{mt}, \mathcal{P}_{mb}, \mathcal{P}_{ml}, \mathcal{P}_{mr} \rangle$ to be an* **fo:simple−page−master** *such that:*

- *height* $= \mathcal{P}_h$ *and width* $= \mathcal{P}_w$*;*
- *margin−top* $= \mathcal{P}_{mt}$ *and margin−bottom* $= \mathcal{P}_{mb}$*;*
- *margin−left* $= \mathcal{P}_{ml}$ *and margin−right* $= \mathcal{P}_{mr}$*.*

Definition 2 (Bottommost layer). *Let $\mathcal{A}$ be the set of all* **fo:region−arbitrary** *elements within a given page master $\mathcal{P}$, each of which containing a distinct layer $= \lambda$ attribute. We define Λ to be the greatest value of λ.*

Definition 3 (FO Regions → Arbitrary). *Standard XSL-FO region elements, i.e.* **fo:region−before**, **fo:region−after**, **fo:region−start**, **fo:region−end** *and* **fo:region−body**, *within a page master $\mathcal{P}$, such that extent $= \varepsilon$ and precedence $= \pi$ (for non-body regions) generate rectangular arbitrary regions with recede, layer, x, y, width and height attributes according to Table 1.*

An important aspect concerning the definitions of the *start* and *end* regions is that the value of the precedence attribute is assumed to be previously validated with regards to the precedence defined for the *before* and *after* regions.

Another aspect of the proposed mapping refers to the way in which the body region is converted into a rectangular area that spans the entire page. In our proposed mapping we take full advantage of the semantics associated to the layer and recede attributes to generate a region located at the bottommost layer, and will recede its area in favor of all the other regions within the page master.

4. AREA TREE MODEL FOR ARBITRARY SHAPES

Besides specifying the syntax for formatting objects, XSL-FO also contains a general area model (area tree) which comprise the formatted result [14], as well as the interaction among these areas. The model is intended to provide an abstract framework which is used in describing the semantics of formatting objects.

4.1 Arbitrary Areas

The main modification introduced into the area model is the introduction of *Arbitrary Areas* as possible nodes within the area tree. Arbitrary areas are hierarchically equivalent to the *Rectangular Areas* defined in Section 4.2 of the XSL-FO 1.1 specification. Like its rectangular counterparts, specialized arbitrary areas can be either block-level or inline-level. Similarly, child elements of block-level arbitrary areas are of any type, either rectangular or arbitrary, inline or block-level, while arbitrary children of inline-level are only inline-level. Unlike arbitrary areas, child elements of rectangular areas can only be rectangular.

4.1.1 Border, Padding and Content

An important set of concepts within the original area tree model involve the abstract naming of rectangle edges (*i.e.* start, end, before, after), which are used as basis for a language-independent representation of areas. Association of such naming to the edges of arbitrary areas is clearly not feasible. We therefore use the minimal bounding box within which an arbitrary area can be inscribed whenever the abstract edges of a rectangular area are required for writing-mode or coordinate orientation purposes (Figure 13).

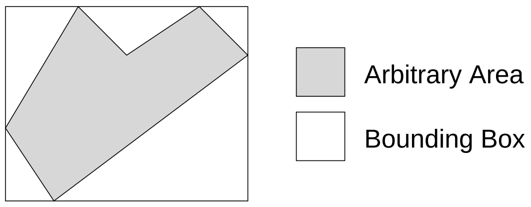

Figure 13: A Bounding Box

Areas in the original XSL-FO area tree had a series of optional enclosing rectangles referring to padding, border and content. Arbitrary areas have no rectangle or other enclosing shape defining padding space, whereas they enclose a content shape which coincides with the outline of the area outline itself. Furthermore, if an arbitrary area has a border property the corresponding border shape is generated using the same path outline as its enclosing arbitrary shape.

Nodes within the area tree have a set of associated traits, which are either directly derived from formatting object (FO) properties or indirectly derived by one or more of such properties. Directly derived traits obey the same semantics as their generating FO properties for arbitrary areas. The majority of indirectly derived traits also keep their original semantics with the exception of the is−reference−area and is−viewport−area traits, and those specifying position and offset modifications. More precisely:

- Arbitrary viewport areas can never be used for scrolling content;
- Whenever an arbitrary area is also a reference area, the coordinate system used by its children is defined by its bounding box;
- Therefore, whenever child elements of arbitrary reference areas contain position or offset traits, these are calculated relative to their parent's bounding box;
- Arbitrary viewport/reference pairs are always coinciding shapes.

Arbitrary areas containing traits that cause child areas to have smaller rectangular bounds than its parent (*e.g.* start-indent, end-indent ...), generate a clipping rectangle derived from its bounding box and resized accordingly, which is then used to clip the resulting arbitrary area. An example of such operation is the clipping of an arbitrary region which exceeds the defined page margins (Figure 14).

Moreover allocating new block areas throughout the layout of a page sequence also uses a similar approach. Namely,

Region	x	y	width	height	recede	layer	
						π	$\neg\pi$
fo:region-before	$\mathcal{P}_{ml}$	$\mathcal{P}_{mt}$	$\mathcal{P}_w - \mathcal{P}_{ml} - \mathcal{P}_{mr}$	ε	$\neg\pi$	$\Lambda + 1$	$\Lambda + 3$
fo:region-after	$\mathcal{P}_{ml}$	$\mathcal{P}_h - \varepsilon - \mathcal{P}_{mb}$	$\mathcal{P}_w - \mathcal{P}_{ml} - \mathcal{P}_{mr}$	ε	$\neg\pi$	$\Lambda + 1$	$\Lambda + 3$
fo:region-start	$\mathcal{P}_{ml}$	$\mathcal{P}_{mt}$	ε	$\mathcal{P}_h - \mathcal{P}_{mt} - \mathcal{P}_{mb}$	$false$	$\Lambda + 2$	$\Lambda + 2$
fo:region-end	$\mathcal{P}_w - \varepsilon + \mathcal{P}_{mr}$	$\mathcal{P}_{mt}$	ε	$\mathcal{P}_h - \mathcal{P}_{mt} - \mathcal{P}_{mb}$	$false$	$\Lambda + 2$	$\Lambda + 2$
fo:region-body	$\mathcal{P}_{ml}$	$\mathcal{P}_{mt}$	$\mathcal{P}_w - \mathcal{P}_{ml} - \mathcal{P}_{mr}$	$\mathcal{P}_h - \mathcal{P}_{mt} - \mathcal{P}_{mb}$	$true$	$\Lambda + 4$	$\Lambda + 4$

Table 1: Region conversion table.

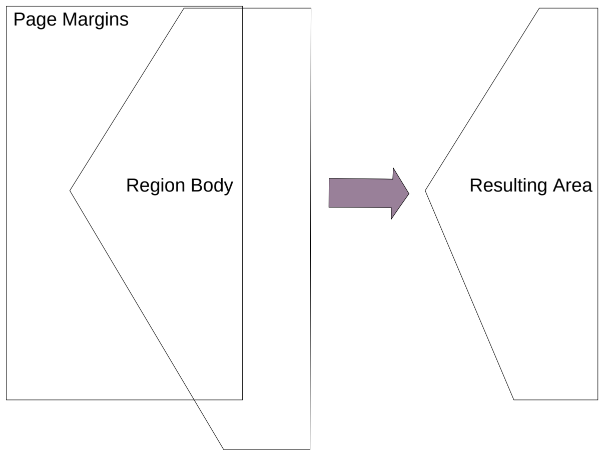

Figure 14: Area Clipping.

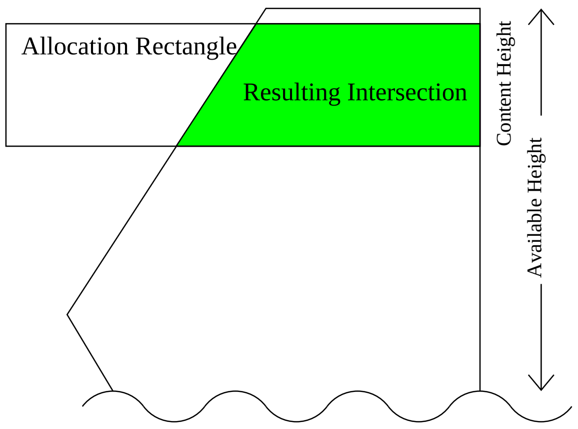

Figure 15: Area Allocation.

the allocation rectangle used to specify the available space for the line building algorithm is used as a clipping box and intersected with the parent arbitrary area (Figure 15).

4.2 Line Building

As the formatter populates the area tree through the refinement of areas into more specialized block areas in order to layout content, it reaches a point in which *Line Areas* are generated. These areas will provide the line building algorithm with information regarding contiguous inline progression space. Such information is used by the formatter to determine the amount of content that will be placed within each line of a given document, and drive the actual layout process.

Arbitrary *Line Areas* generate inline areas using the same strategy proposed by SVG 1.2 to calculate text spans [15]. More precisely, line areas are generated as would any other block-level areas, as described in Section 4.1.1, resulting in zero or more closed shapes within which content is to be laid out. Following this process, the layout algorithm must determine which shapes are actually within the parent line area. To accomplish that, the algorithm described in [15] is used, with the following modifications:

- The areas that would generate text regions within SVG flowing text layout will generate content-bearing inline child elements;
- All the remaining SVG spans will generate inline spaces within an arbitrary line area.

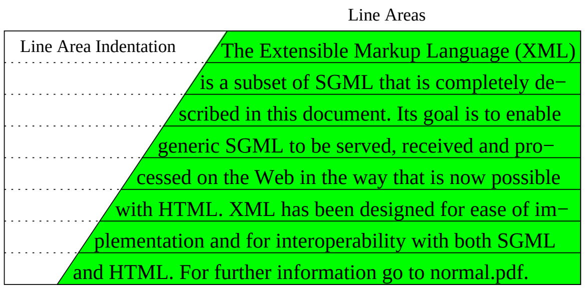
Line Areas
Line Area Indentation
The Extensible Markup Language (XML)
is a subset of SGML that is completely de−
scribed in this document. Its goal is to enable
generic SGML to be served, received and pro−
cessed on the Web in the way that is now possible
with HTML. XML has been designed for ease of im−
plementation and for interoperability with both SGML
and HTML. For further information go to normal.pdf.

Figure 16: Line Areas and Indentation.

Some traits within the resulting areas are modified according to the alignment and spacing traits defined for its parent areas. Considering a line area L contained within a block area B, an inline area I is modified as follows:

- Start indentation for the first I generated for L is modified by the addition of the space from the start-edge of content allocation rectangle for B to the start edge of L (Figure 16);
- End indentation for the last I generated for L is modified by the addition of the space from the end-edge of content allocation rectangle for B to the end edge of L;
- The spacing introduced to enforce a justified alignment for a given line area is generated individually for each inline element I generated for L instead of for every inline element of L.

5. IMPLEMENTATION

Laying out content in multiple arbitrary regions is by no means a computationally trivial operation. One of the main concerns regarding the proposed extensions is the feasibility of handling multiple arbitrary areas in the source XSL-FO, as well subsequently breaking down these areas in order to generate the appropriate sub-areas. A solution to such question can be provided through an implementation of an arbitrary shape processor for XSL-FO.

Implementing a full-fledged Formatting Objects Processor is, nevertheless, a complex task to the point that there is no open-source implementation of the entire XSL-FO standard. Furthermore, implementing a limited subset of the standard plus our extensions would also be a very demanding implementation effort. We therefore chose to modify an existing implementation to include our extensions, namely Apache Formatting Objects Processor (FOP) [5]. These extensions are summarized in Section 5.1, while examples of extended FO documents and the associated results using the proposed implementation are described in Section 5.2.

5.1 Extended Formatting Objects Processor

Apache FOP is generally recognized as the leading open-source implementation of the XSL-FO Standard [6, 9], and was therefore chosen as the basis for the prototype implementation. Despite of the prototype being developed using a specific implementation as its starting point, we will attempt to describe the required modifications as generically as possible, in order to allow the requirements to be used in the extension of other XSL-FO implementations.

Considering the usage of features from XSL-FO 1.1 and the new XML elements used by the extension, a 1.0-compliant processor has to have its parser module modified to include handling of the following elements:

- **fo:flow−map** - to handle the mapping of content-flows into its destination regions, as well as its child element **fo:flow−assignment**, and its associated flow−map−name attribute. Also, the **fo:page−sequence** element will have to be augmented with the flow−map−reference attribute to allow a page sequence to select which flow map it will use in the layout process;

- **fo:flow−assignment** - which holds the association of flows to regions;

- **fo:flow−source−list** - which specifies a list of flows that are associated to regions in the current simple page master;

- **fo:flow−name−specifier** - and its associated flow−name−reference attribute;

- **fo:flow−target−list** - which specifies the list of corresponding regions into which the flows specified in the **fo:flow−source−list** will be laid out;

- **fo:region−name−specifier** - and its associated region−name−reference attribute;

- **fo:region−arbitrary** - to describe the new arbitrary regions that can be used in a document layout, as well as its associated recede and layer attributes.

These new elements also require their corresponding class encoding to handle their associated semantics, according to the XSL-FO 1.1 specification, or to Section 3 for arbitrary regions. Furthermore, the possibility of laying out multiple content flows in the same page sequence requires the layout process to be modified to handle the page-by-page distribution of parallel content flows.

The usage of a non-rectangular area representation entails the usage of an encoding capable of keeping track of all the edges of arbitrary areas. Therefore, the encoding of geometric descriptions within an implementation of arbitrary shapes is greatly improved when decoupled from the representation of area tree objects, in order to allow the Inline or Block-level behavior to be handled separately from the geometric calculations. The prototype thus uses Java2D Area [11] objects to store the geometric definitions for any given element in the area tree. Such an encoding also allows the prototype to take advantage of Java constructive geometry API. In addition to these facilities, the selected object representation can be generated from the SVG shape description in a straightforward way using the Batik [4] toolkit.

Finally, the modified area tree elements must be rendered into an assortment of target formats and media, which entails modifications into the XSL rendering process. The usage of Java2D objects is also advantageous in this situation, as many Java-based XSL rendering APIs rely on instances of the Graphics2D class, for which many specialized implementations intended target formats such as PDF, PostScript, SVG or JPEG are available.

5.2 Results and Limitations

One of the main new capabilities included in the prototype is the possibility to define an arbitrarily-shaped area for the body region within a page layout. This allows flow content to be laid out within this arbitrary shape across multiple pages. The shapes associated to the body region are currently specified using SVG [16]. The specified SVG is read by the parser contained in FOP, and the outline of the resulting picture defines the boundaries for the arbitrary area. The prototype uses the latest Batik library to parse such SVG and convert it into Java2D objects, which are then used to derive the area shape. For example, an XSL-FO definition within a simple page master defining an yin-yang shaped area on top of rectangular area would generate pages looking like Figure 17.

By using Java2D to perform area calculations, some overhead is introduced into the implementation. This is due to the usage of absolute-positioned Shape objects rather than integer pairs defining the size of the generated areas. Since area intersection calculations scale on the number of vertices involved in the computation, one should expect a higher cost for very complex page regions. We performed a series of benchmarks in order to assess the overhead of processing XSL-FO 1.0 documents in the new calculation model, which showed that the new calculation model scales linearly with regards to the original calculations (Figure 18).

6. CONCLUDING REMARKS

This paper describes a proposal for the extension of the XSL-FO standard to be able to layout content into multiple arbitrarily shaped page regions. The extension is intended

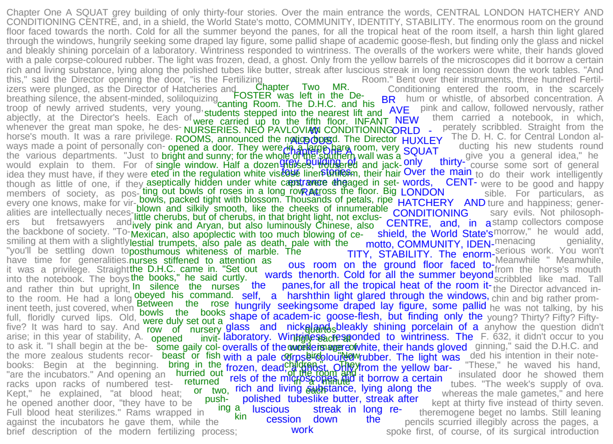

Figure 17: Content laid out over yin-yang symbol.

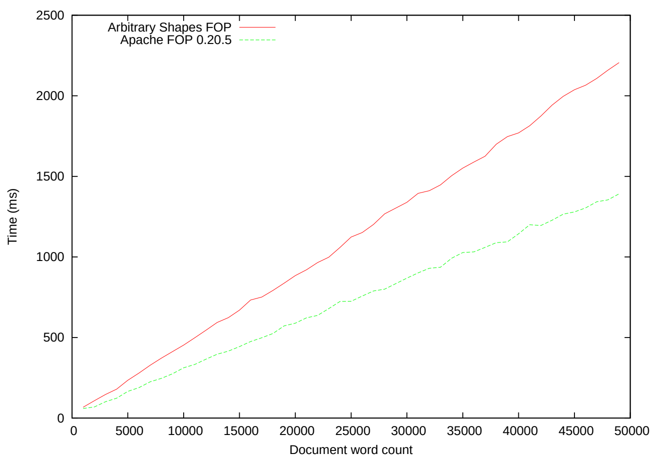

Figure 18: AS-FOP versus Apache FOP.

to leverage the XSL standard for usage in the typesetting of complex document layouts usually found only in expensive desktop publishing applications. An implementation of such standard enables the generation of personalized documents in layouts only encountered in one-of-a-kind graphic presentations.

The proposed extensions reuse many concepts from the SVG standard, as well as concepts currently being proposed for its upcoming 1.2 version. Such reuse is not incidental, as ensuing implementations of the arbitrary layout algorithms and shape encoding schemes will be able to share many common components. At some level this sharing of components is currently being done in the implementation of FOP, as well as in the extended prototype described in Section 5.

Even with the text-flow enhancements incorporated into the 1.2 version of SVG, we believe that the two standards have distinct purposes. SVG is a language for defining two-dimensional graphics, whereas XSL-FO comprises a stylesheet and a document formatting language. The authors believe that while SVG is an appropriate language for defining the geometric constructs in an FO input, as well as a complete output format for rendered content, SVG should remain responsible for the quick visualization of 2D content without performing complex typesetting processing. XSL-FO should instead be used for batch processing large volumes of data into formatted and paginated content.

The next step in the development of the arbitrary shapes extensions of XSL-FO is its analysis by the community. Such an analysis should uncover potential flaws in the specification or point out improvements, and potentially lead to its incorporation into a future version of the standard.

7. REFERENCES

[1] ADOBE® SYSTEMS. *Postscript language tutorial and cookbook*. Addison-Wesley, 1985.

[2] ADOBE® SYSTEMS. *PostScript™ Language Reference Manual*, 2nd ed. Adobe Systems Incorporated, 1990.

[3] ADOBE® SYSTEMS. *PDF Reference*, 4th ed. Adobe Systems Incorporated, 2003.

[4] APACHE SOFTWARE FOUNDATION. Batik SVG Toolkit. Web Page, September 2004. Extracted from http://xml.apache.org/batik/.

[5] APACHE SOFTWARE FOUNDATION. Formatting Objects Processor. Web Page, September 2004. Extracted from http://xml.apache.org/fop/.

[6] CANFORA, G., AND CERULO, L. A visual approach to define xml to fo transformations. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (2002), ACM Press, pp. 563–570.

[7] KNUTH, D. E. *The TEXbook*, vol. A of *Computers & Typesetting*. Addison-Wesley, 1986.

[8] KREULICH, K. Publishing Workflows with XSL-FO. In *XML Europe 2003* (London, England, 2003), International Digital Enterprise Alliance, pp. 1–6.

[9] PAWSON, D. *XSL-FO: Making XML Look Good in Print*. O'Reilly, United States, 2002.

[10] SHIRLEY, P. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 2002.

[11] SUN MICROSYSTEMS INC. JavaTM 2 platform, standard edition, v 1.4.2 API specification. Website, September 2004. Extracted from http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[12] TEX USERS GROUP. Comprehensive TEX Archive Network (CTAN). Extracted from `http://www.tug.org/ctan.html`, 2004.

[13] W3C, WORLD WIDE WEB CONSORTIUM. Extensible Stylesheet Language (XSL) Version 1.0. Web, October 2001. Extracted from http://www.w3.org/TR/xsl/.

[14] W3C, WORLD WIDE WEB CONSORTIUM. Extensible Stylesheet Language (XSL) Version 1.1. W3C Working Draft, December 2003. Extracted from http://www.w3.org/TR/2003/WD-xsl11-20031217/.

[15] W3C, WORLD WIDE WEB CONSORTIUM. Scalable Vector Graphics (SVG) 1.2. W3C Working Draft, May 2004. Extracted from http://www.w3.org/TR/SVG12/.

[16] W3C,WORLD WIDE WEB CONSORTIUM . Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation, January 2003. Extracted from http://www.w3.org/TR/SVG11/.