Design of an Efficient Scalable Vector Graphics Player for Constrained Devices

Cyril Concolato, J. Le Feuvre and J.-C. Moissinac

Abstract — *The mobile industry, and in particular the 3rd Generation Partnership Project (3GPP) Consortium, has selected the "Tiny" profile of the Scalable Vector Graphics (SVG) specification as a basis for the Rich Media format for mobile applications, leading the way to consumer electronics. Among the foreseen applications for SVG on constrained consumer electronics devices, maps, clip arts, animated cartoons and user interfaces are often cited. However, such devices are memory-constrained and have limited processing capabilities whereas vector graphics clip arts, animated cartoons or maps are described by large SVG documents. Such content may require heavy computations and important memory consumption, especially when applying models for animation and inheritance. In this paper, we present the design of a low-footprint and computationally efficient player for large and animated SVG documents. We describe the structures of the scene objects which enable low memory consumption and the compositing and rendering algorithms enabling fast playback. Finally, we evaluate the limitations of our proposal and compare our results with publically available desktop SVG players.*[1]

Index Terms — Design, Multimedia Scenes, Scalable Vector Graphics, Visualization.

I. INTRODUCTION

Advances in networking allow today the distribution of multimedia services to a wide range of devices, including constrained devices like PDAs or mobile phones. Such services are dynamic, interactive collections of multimedia data such as audio, video, graphics, and text. They range from movies enriched with vector graphic overlays and interactivity to complex multi-step services with fluid interaction and different media types and are often called rich-media services. Following these advances, many standardization activities have started to specify and/or recommend formats adequate for both these services and these new devices. Proprietary formats have also been developed or adapted for this purpose. The standard format which has been selected by the mobile industry, namely by the 3rd Generation Project Partnership (3GPP), is SVG.

Scalable Vector Graphics (SVG) [1] is a recommendation from the World Wide Web Consortium (W3C) that defines, in its first version (1.0), an XML language to represent interactive and animated vector graphics. It builds on other W3C recommendations, like the Synchronized Media Integration Language (SMIL) [2] for the description of the timing and animation behavior or the Cascading Style Sheet (CSS) [3] specification for the styling feature. Version 1.2 of SVG is currently being specified. It includes SMIL media elements (i.e. audio, video and animation) as well as other improvements (e.g. text layout, navigation behavior) which make SVG a complete multimedia scene description language. Its “Tiny” profile defines the basis for the Dynamic and Interactive Multimedia Scenes (DIMS) [10] format mandated for the mobile phones in Europe.

According to a recent press release, more than 375 millions mobile phones are equipped with an SVG player [9]. Among the foreseen applications for SVG on consumer electronics, maps, clip arts, animated cartoons, games and user interfaces are often cited. Despite the high amount of standardization activities in the area and the existence of commercial products [9][11], playback of large animated content on constrained end-user devices remains a challenge, especially in term of memory consumption and surprisingly little research work has been done in this area.

This paper proposes an innovative design of an SVG player which allows reaching an interesting compromise between memory consumption and playback speed. This design is based on tailored scene structures and on accompanying optimized algorithms for compositing and rendering.

The rest of this paper is organized as follows. Section 2 describes the generic process of Multimedia Visualization. Section 3 gives some further considerations about the design of an efficient multimedia player, including SVG specifics. Section 4 details the proposed structures and algorithms. Section 5 presents the results achieved with this proposal. Finally, Section 6 concludes this paper and presents future work.

II. MULTIMEDIA VISUALIZATION PROCESS

The multimedia scenes visualization process can be viewed as a cyclic three-step process, as depicted in Figure 1 and described in the following.

[1] Manuscript received April 15, 2008. This work was supported by the European Commission under the INTERMEDIA Network of Excellence.

C. Concolato, J. Le Feuvre and J.-C. Moissinac are with TELECOM ParisTech, Paris, France. (e-mail: concolato@enst.fr, lefeuvre@enst.fr and moissinac@enst.fr).

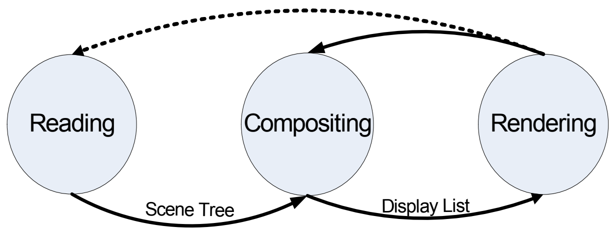

Figure 1 – The visualization of an animated and interactive multimedia scene is a three-step cyclic process: Reading, Compositing, and Rendering. Reading happens first and may happen only once or several times as new data arrives. Compositing happens after Reading but it may happen more often than Reading if required by animations, user interactions or synchronization of media streams. Rendering happens last, each time after the Compositing step, to produce the visual and aural result.

A. Reading

The first step in the visualization process consists in reading scene description data, from a textual source (XML or not) or from a binary source, and in producing a memory representation of the scene objects suitable for the Compositing step. This memory representation is often referred to a scene graph or a scene tree. The source of data to build this scene tree may be a file or a stream.

On the one hand, if the source of data is a file, depending on the reading algorithm, on the file size, on the location of the file (local or remote) and on the throughput of the disk or network, the entire scene may be read in one time. In this case, the scene objects transmitted to the Compositing step represents all the information required during the lifetime of the scene.

On the other hand, if the source of data is a stream or if it is a file read progressively, the Reading step may be called several times to update the set of scene objects shared with the Compositing step. These updates may be addition or deletion of scene objects, or modification of the properties of a scene object. These modifications may result from the reception of new additional untimed data or of timed scene modifications like MPEG-4 LASeR [5] updates or events as specified in “Remote Events for XML” (REX) [12], or from the use of programmatic interfaces such as the XMLHttpRequest object [13].

In this paper, we will not assume one or the other means for processing the data. In particular, in the design of the player, we will not make assumptions on the data not changing after the first Reading step. In other words, we will not use methods such as analyzing the whole scene to determine some properties of the scene and make specific optimizations. Additionally, this paper will not focus on improving the Reading speed as lots of research has been done on this topic already, either by working on the design of precompiled XML parser [6] or by defining binary formats faster to parse [5][17]. The focus of this paper with respect to the Reading step will rather be on the design of the scene objects, produced while reading, to reach the optimum memory consumption within the constraints of the Compositing step.

B. Compositing

The Compositing step is an additional step compared to the two-steps traditional audio/video visualization process which usually consists in decoding and synchronously rendering. This Compositing step is required for several reasons in the generic multimedia visualization process. A first reason is animations. Indeed, multimedia scenes may be animated and animations need to be timely updated, to reach the desired visualization frame rate and the desired smoothness of presentation, even if no new data is received from the Reading step. A second reason is interactions. True multimedia content includes interactions. For example, the user may navigate in the scene or may trigger complex behaviors. These interactions result in modifications which need to be applied to the scene objects before rendering and independently from the Reading step.

The Compositing step therefore consists, at each visualization cycle, in traversing the scene objects to update animations and apply the modifications resulting from user interactions. The result of this traversal is a set of ready-to-be-rendered scene objects, also called Display List or Graphics List.

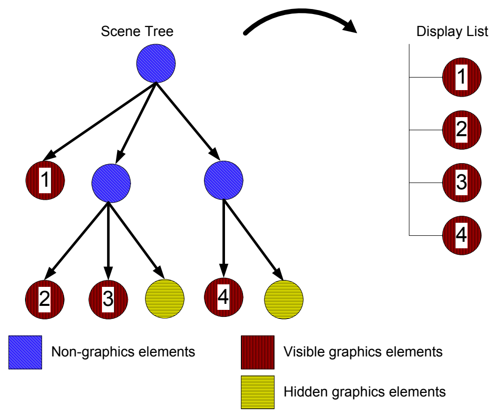

Figure 2 – The Compositing step produces a Display List based on the visible graphics element present in the Scene Tree.

It is important to note that the scene structures shared between the Compositing and Rendering steps are not necessarily the same as the objects shared between the Reading and the Compositing steps. In particular, the scene objects required for compositing need to provide mechanisms for animations and for user interactions (e.g. read/write access for scripting), whereas the scene objects required for rendering need to be adequate for video and audio rendering hardware/software interfaces such as OpenGL or OpenVG [14]. These compositing and rendering requirements may be conflicting, in which case different representations are used. For example, the rendering of a Bézier curve may imply producing a list of line segments while performing animation of this curve may require accessing the curve control points. Additionally, because the animations and interactions may mute or make invisible some scene objects, not all objects manipulated by the Compositing step are forwarded to the Rendering step, as depicted in Figure 2.

We can see from this description that the performance of the Compositing step may be impacted by the structure of the scene (i.e. how many objects need to be traversed to compose the scene) and by the structure of each object (i.e. how easy it is to access the object properties). In this paper, we propose a design for scene objects together with compositing algorithms compatible with the theoretical SVG compositing model.

C. Rendering

The last step in the visualization cycle is the Rendering step. It consists in producing, from the scene structures as forwarded by the Compositing step, the visual and aural result for the end-user. In this paper, we will concentrate only on the visual part of the rendering and omit the audio part.

To achieve efficient rendering, many existing algorithms use indirect rendering where only the parts of the screen that have changed are refreshed from one frame to another. There are several reasons why a part of the screen may change: either the geometry of an object has changed (e.g. if the length of a rectangle is animated), or its position has changed, or finally, its appearance (fill or stroke). Additionally, if an object has changed, it may be necessary to redraw (parts of) other objects due to transparency or anti-aliasing. We consider that an efficient rendering algorithm should minimize the computations required for detecting changes. Though the rendering of the object can be language independent, the change detection is specific. In this paper, we will present an algorithm to detect changes, optimized for SVG content.

III. Design Considerations

In the previous section, we have explained the generic process of multimedia visualization, highlighting where our contribution will reside. In this section, we present some design consideration that must be taken into account to reach our goal of designing a player that is computationally efficient and that consumes the minimum amount of memory. We start by presenting first some general considerations and then we give some SVG specific ones.

A. General Considerations

As we have seen, the role of the Reading step is to produce a memory representation of the scene that is efficient for the Compositing step. To design a memory efficient scene representation, it is important to understand the following three generic aspects about multimedia scenes.

Multimedia scenes are made of a collection of primitives. These primitives may be graphical primitives (rectangle, curves, text …), media primitives (images, audio, and video), structuring primitives (groups), and primitives for animation or interactivity. Their complexity may range from a simple audio/video scene to highly sophisticated maps, cliparts, animated cartoons or user interfaces. In this paper, we will focus our effort on the most frequent and largest scenes, i.e. those using many graphical objects and animations.

In the design of a memory-efficient representation of scene objects, we must also take into account the easiness for accessing the data during the Compositing step. For example, one could decide to store the objects in a compressed form, the memory consumption would be minimal but each read access during the Compositing step would require decoding the data, and possibly encoding it when the object is modified by scripting or animation. Such a design would therefore be unacceptable, especially for heavily animated content, like cartoons or games, where a high frame rate is desirable.

Finally, the design of the memory representation of scene objects for compositing may be different from the one used for creating the scene in an authoring tool, or from the one used to transmit the scene. However, it should not be too far from the one defined by the specification that describes the language. Indeed, rich media services more and more rely on scripting to modify the scene, such as in Ajax applications [15]. The problem is that scripts use specific programming interfaces which should not be hindered by the design choice.

B. SVG Specific Considerations

There are three aspects of the SVG language which have important consequences on the design of a player that we would like to highlight here: SVG scripting, SVG and CSS inheritance; and SMIL animation and CSS inheritance.

First, as we have explained, the design of SVG scene objects shall be compatible with and efficient for scripts. With SVG scenes, a content creator may use either Document Object Model (DOM) [4] interfaces or MicroDOM [1] interfaces to modify a scene by scripts. MicroDOM interfaces have been designed for mobile devices relying on typed data instead of strings, simplifying navigation in the tree, etc. Our algorithms and structures should therefore rather be compatible with Micro-DOM than with DOM interfaces.

Secondly, like most scene description formats, SVG content uses a tree structure. Nodes in this scene tree are grouped according to their spatial properties. However, one important aspect of the SVG language, which impacts compositing and rendering, is its integration with web technologies like the Cascading Style Sheet (CSS) specification. Specifically, SVG reuses, from that specification, the concept of property inheritance. According to this concept, some SVG attributes, called presentation attributes, which actually correspond to CSS properties, may be specified on grouping nodes. In this case, their values (possibly after some intermediate computation) are forwarded to the children of the grouping node. Consequently, a child node may inherit the properties of its parent node in the scene tree. This behavior also exists in HTML or XHTML and allows applying a common style to a whole scene subtree. This impacts greatly the design of the scene elements and the memory requirements. On this aspect, we found, in research papers, the work of Cogliati and Vuorimaa in [7] which deals with the design of an optimized Cascading Style Sheet engine with memory constraints. This work focuses on the integration of a CSS engine in generic XML browser but does not address the efficient design of the scene objects, and especially not for SVG elements.

Finally, the last specific aspect of SVG compared to other scene description languages lies in its animation model. The SVG animation model follows the model defined by the Synchronized Multimedia Integration Language (SMIL), which defines the notion of base and animated value. Therefore, theoretically, scene objects should maintain two values per animatable attributes. Additionally, according to the SMIL so-called "Sandwich model", when CSS is used in conjunction with the animation model, three values per animated attributes should be accessible (see Figure 3). In this paper, we propose a design which allows for a memory efficient representation of this model.

IV. PROPOSED DESIGN AND ALGORITHMS

A. Scene Objects Structures

Our first design choice is to store attribute values as typed data as opposed to strings. This choice has two impacts. First, the Reading step is slowed down because attribute value parsing is required but, the Compositing step is fasten because accessing typed data is obviously faster. Since compositing happens more often than reading, on the overall, we believe this choice to be positive. This impacts as well the scripting performance when string values are used, which affects only DOM scripts, since MicroDOM scripts uses typed accesses. This choice is therefore consistent with our objectives to enable efficient playback on constrained devices and to stay close to the MicroDOM design.

The second choice concerns the structure of the scene objects. Two approaches were possible. The first one consists in creating a node structure different for each type of node and with all the possible attributes for this type statically allocated. This has the advantage of a fast allocation process and a fast access to the attributes values. The second one consists in allocating a generic structure for all types of nodes and in which the attributes are allocated only when there are specified in the input source. When experimenting with SVG, it appears rather rapidly that the first approach is not very optimal. Indeed, in SVG, due to inheritance, many attributes can be potentially specified on many elements. For example, the SVG "rect" element can have up to 67 attributes. Hence, using the first type of structure would consume an unnecessary amount of memory. We therefore went for the second option as described in the code below.

```
struct SVGAttribute {
   int attribute_identifier;
   void *value;
};

struct SVGElement {
   int type;
   ListOf(SVGAttribute) attributes;
   ListOf(SVGElement) children;
};
```

This second option also offers the advantage of being compatible with DOM/MicroDOM APIs which require the possibility to delete an attribute or to tell if the attribute was specified or not in the source of data. The drawback of this method is that, if no care is taken, it requires iteration of the list of attributes each time the access to an attribute is needed. This is the case for example when accessing the width, then the height, then the top-left position then the color of a rectangle. To avoid these successive iterations, we designed an additional data structure which allows accessing any attribute with a single iteration of the list of attributes. Such structure, described in a simplified form in the code below, is created only when needed, filled with the list of specified attributes, and discarded when no longer needed. This structure currently contains pointers for the approximately 200 possible attributes in SVG Tiny 1.2, as illustrated below.

```
struct SVGAllAttributes {
  SVG_ID *id;
  …
  SVG_Paint *fill;
  …
  SVG_Coordinate *x, *y;
  SVG_Length *width, *height;
  …
  SVG_Transform *transform;
  …
};
```

B. Compositing Algorithm

In the previous section we have presented the basic structures of the SVG scene tree that we will use. We now present the associated compositing algorithm.

The Compositing step requires that, at each visualization cycle, the scene tree is traversed to determine the visual parameters of each visible object. This means that some user events need to be processed, timing dependencies resolved, animations applied and the spatial positioning of each visual element computed. In order to present a fluid visualization of the scene, the Compositing step must be short and therefore each of these sub-steps must be optimized.

The requirements of our algorithm are three-folds: to perform at most one traversal of the scene tree during one visualization cycle; to limit the number of nodes being traversed at each cycle; and, to limit the number of operations performed for each node.

Our approach considers that compositing an SVG scene can be divided in three separate processes:

- Handling of the temporal primitives, including media primitives (video),
- Handling of the event related primitives, and in particular, of script elements,
- And finally, handling of the graphics and layout primitives.

This division of the general compositing algorithm has the following advantage. If, after processing the first two steps, we detect that the graphical objects do not need to be processed (because no change has happened since the previous cycle), the compositing stops without traversing the major part of the scene tree. This allows for reducing the compositing time of large scenes which contain only few animations or few interaction primitives.

1) Handling of timed elements

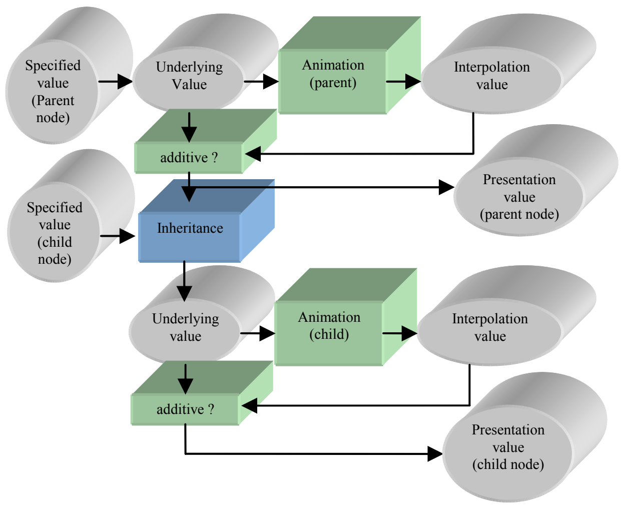

Figure 3 – In the SMIL Animation Sandwich model, the result of the animation of a node may depend on an animation at the parent level in the scene tree.

Due to the inheritance and animation sandwich model, it is not obvious that the proposed division of the compositing process can be made while keeping the result conformant with the SVG specification. In particular, the computation of the presentation value of an animation may depend on the actual position of the animation element in the scene tree. For instance, if an animation element, animating a presentation attribute, adds its interpolation value to the inherited value of this property, then the resulting presentation value will also depend on the presentation value at a parent level in the scene tree. This behavior is illustrated in Figure 3.

To optimize the animation process, we first note that performing animations involves first determining if the animation is active, then computing the parameters of the animation (begin, end, duration, fraction of the animation duration, interpolation coefficient), and finally computing the interpolation value and modifying the scene. Additionally, we also note that the SVG specification defines only one time line per document and that there is no relationship between the CSS inheritance mechanism and the SMIL timing model. Consequently, the first two tasks (determining the activation, and computing the animation parameters) can be performed outside of the main tree traversal, independently from the tree traversal.

Based on this separation, our algorithm uses a flat list (as opposed to a tree representation) of the timed elements, stored at the compositor level. This list is traversed more easily, before and independently of the main tree traversal to notify the new scene time to the timed elements and to resolve timing dependencies among them. Additionally, in our algorithm, we also note that for timed media elements (i.e. video), the final processing of the element, that is to say the synchronization of the output of the media decoders with the scene, which does not interfere with CSS, can also be applied before the processing of the scene tree.

2) Handling of user events

Concerning user events, according to the interactivity model as defined by the DOM Events Processing Model [4] and as reused in SVG, several types of elements need to be processed: listener elements, which specify that the capture for specific user events is required; observer elements on which the event is actually observed; target elements on which the event is targeted; and handler or script elements which react to the actual occurrence of the event after propagation of the event in the scene tree.

Our approach is here similar to the one applied for animation. We want to be able to handle events-related elements independently from the graphics scene tree. To that purpose, we can note that the processing of listener elements does not depend on the positioning of the element in the scene tree. However, because of the bubbling and capture phase of the DOM event processing model, the handling of the events by scripts or handlers elements requires the propagation of the event in the tree from the observer to the target. But we can also note that this propagation can be made without any relationships to the CSS inheritance model. Consequently, our design uses a list of listener elements, stored in the target element, which is traversed independently from the main tree traversal. A consequence of this algorithm is that in order to perform the bubbling or capture phase, each node in the scene tree must also contain a link to its parent node.

3) Handling of graphics and layout elements

At this stage, we have described that the scene tree composition comprises a first step for notifying the time to the timed elements and a second step for handling user events. We describe here the algorithm for the last step which is the traversal of a scene tree in which only the graphical and layout elements remain to be processed.

A particular difficulty that needs to be solved in this last part is the one of the theoretical animation sandwich model which requires the use of a base value, of a computed value and of a presentation value. Indeed, keeping these three values for each attribute of each element would consume an unnecessary high amount of memory. To solve this problem, we note that the base value needs to be kept only for the attributes which are actually animated, and if we also note that only the attributes which correspond to CSS properties need to maintain the notion of computed value; then we can derive the following proposal.

We propose, in each node, to store a list of animations which apply to this node. In this list, animations are grouped according to the attribute they target. Consequently, we store the base value of an animated attribute in the animation group itself. Hence, our algorithm does not duplicate the memory consumption for all attributes, but only for animated attributes, with only one copy of the base value regardless of the number of animations.

With respect to the CSS computed value, our algorithm leverages the recursive characteristic of the scene tree traversal to store temporarily the computed value of each CSS property. More precisely, we define a property context. This context is a collection of pointers to the property value that applies when the context is used. It is initialized at the document level with pointers to the so-called initial values for all possible property. It is then forwarded down the tree during the traversal and modified locally according to the CSS inheritance mechanism: either the property is inherited and the context is not modified, or the property is specified (with a value different from *inherit*) and the context is updated to point to the specified attribute. Additionally, the property context is backed up before applying inheritance and restored before returning to the parent level in the scene tree. Hence, with our algorithm, the compositing of an element at a depth *p* in a scene tree made of *N* elements will consume at most *p+1* property contexts as opposed to *N+1* in a theoretical implementation. The memory consumption is linear with the maximum depth of the tree and is not affected by the number of elements, which is an important advantage for scenes like maps or cliparts. For this description to be complete, we need to indicate that the inheritance and animation processes must actually be mixed. Indeed, when animating, there are cases where interpolation will require using the value of a property of the parent element and producing the result for the current element which will then be used for inheritance to child elements.

The complete algorithm is illustrated in the code below.

```
CompositingStep() {
 Until a stable state is reached, do {
Traverse the list of timed elements;
Evaluate the time attributes;
Determine the animation parameters;
Trigger the begin/end/repeat events;
 }
 For each user event, pick the target {
  For each listener element, do {
   If the event matches {
    Activate the corresponding handler;
    Apply propagation;
   }
  }
 }
 For each media element {
  Synchronize the output of the decoders.
 }
 TraverseElement(root, Initial Context).
}

TraverseElement(SVGElement E,
                PropertyContext C) {
 Backup the Property Context;

 For each animation A targeting E {
  if first animation and first cycle,
    Save the base value;
  if key values use inherit,
    Apply inheritance using C;
  if animation is terminated,
    Restore the base value;
  otherwise {
    Compute the interpolation value;
    Overwrite the presentation value;
  }
 }
 For each property P,
  If E.P != inherit, modify C to point
                             to E.P;

 If visible node,
  Add an object to the display list.

 For each child node E',
   TraverseElement(E', C);

 Restore the Property Context C;
}
```

C. *Rendering Algorithm*

The rendering algorithm we propose works on a display list of SVG graphical elements produced by the Composition step. As introduced in the previous sections, efficient rendering algorithms rely on detecting changes between cycles. We present in this section our method for efficiently detecting changes in SVG scenes.

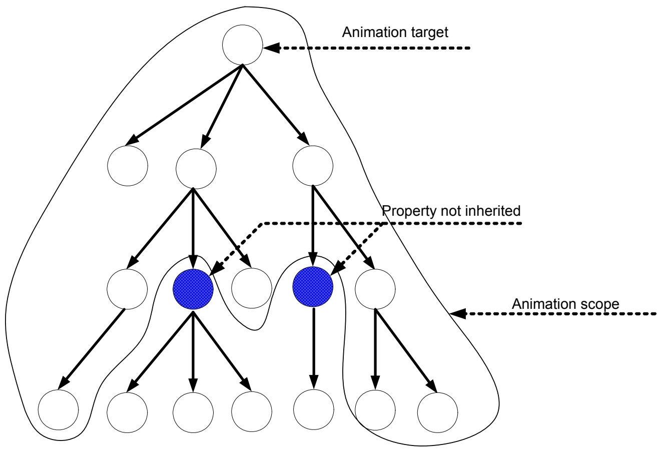

Figure 4 – The scope of the animation of a property in a scene tree is impacted by inheritance and attributes explicitly specified.

There are two sources of changes in SVG scenes: animations or scripts. In our implementation, we also consider LASeR updates as a potential source of updates. In all cases, detecting and propagating changes in an SVG scene is difficult. This difficulty is a consequence of the use of inheritance. Indeed, as we have seen, an animation may modify a property of a grouping element. However, the result of this animation may not apply to the whole sub-tree if some parts of the sub-tree do not inherit this property. This is illustrated in Figure 4.

Additionally, another challenge when detecting changes is to ensure that this detection is also valid for use elements, which are, as specified in SVG, *live clones* of the referencing element. This live cloning implies in particular that change detection shall be made on the use element and not on the referenced element.

Our algorithm first detects when animations are active, and when they produce a different result compared to the previous frame. This detection is not made by comparing the actual result of this animation because this would need actually doing the interpolations and comparing the previous and current value. This would imply unnecessary computations and memory usage, e.g. on complex values like paths. Instead, we detect changes in the result of accumulated animations based on comparisons between previous and current animation parameters (interpolation coefficients). Our algorithm also includes the handling of cumulative and additive animations.

Once we have determined that a combination of animations targeting a same attribute has produced a different result compared to the previous frame, we mark the node as dirty. A node can be dirty in different manners. Several features may have changed: its geometry, its stroke width, its line style … We would therefore need a marker for each feature. In order to keep a compact representation of the markers, using a 32 bits word, we chose to group some features together. This may cause some inefficient redrawing operations in some complex cases but we deem them infrequent enough. The list of markers (less than 32) that we use is given in Table I.

TABLE I
LIST OF SVG SCENE DETECTION CHANGE MARKERS

COLOR_DIRTY	DISPLAYALIGN_DIRTY
FILL_DIRTY	FILLOPACITY_DIRTY
FILLRULE_DIRTY	FONTFAMILY_DIRTY
FONTSIZE_DIRTY	FONTSTYLE_DIRTY
FONTVARIANT_DIRTY	FONTWEIGHT_DIRTY
LINEINCREMENT_DIRTY	OPACITY_DIRTY
SOLID_DIRTY	STOP_DIRTY
STROKE_DIRTY	STROKEDASHARRAY_DIRTY
STROKEDASHOFFSET_DIRTY	STROKELINECAP_DIRTY
STROKELINEJOIN_DIRTY	STROKEMITERLIMIT_DIRTY
STROKEOPACITY_DIRTY	STROKEWIDTH_DIRTY
TEXTPOSITION_DIRTY	VECTOREFFECT_DIRTY
XLINK_HREF_DIRTY	

As we explained, the difficulty of the detection change is due to the use of inheritance together with animations. To solve this problem, we propose to add a novel step to our implementation. We add to the joint inheritance-animation process, described in previous section, a marker inheritance step. In other words, when an animation modifies a property at some level in the scene tree, the target element is marked as dirty for this property and this marker is forwarded to the children node together with the property context. If a child node does not inherit a property, it forces the marker as non-dirty for this property. But if it inherits the property, it will be marked.

V. Results

In previous sections we exposed a set of structures and algorithms for the playback of SVG content. In this section, we first describe the experimental setup that we used to evaluate the performances, then we present the test sequences and we finally give measurements and comparison of the memory consumption and computational efficiency of our method with traditional players.

A. Experimental Setup

We have implemented the proposed structures and algorithms using the C language in the Osmo4 player of the GPAC Framework [8]. For the Reading step, we implemented a SAX parser which is capable of reading SVG Tiny 1.2 documents and which produces scene trees as described in previous sections. We have also implemented the binary decoding of MPEG-4 LASeR streams which also produces the same scene tree. The player, including the proposed algorithm, has been ported on different operating systems for desktop (Windows, Linux) and mobile platforms including Windows Mobile 5.

B. Test Content

In order to evaluate our methods, we needed to evaluate two criteria: memory consumption and computation efficiency. Therefore, we used two kinds of test content: complex and large static vector graphics like maps or clip arts and highly animated graphics. For clip arts or maps, we mostly used SVG maps publically available from Wikipedia. Figure 5 (a) shows an example of such type of content. For animated content, we used content from the SVG Tiny Competition as shown in Figure 5 (b), some of the SVG conformance tests and some cartoons translated from the Adobe Flash format.

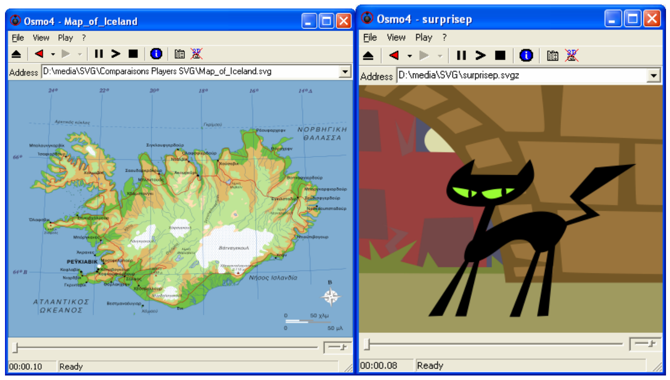
Osmo4 - Map_of_Iceland
File View Play ?
Address D:\media\SVG\Comparaisons Players SVG\Map_of_Iceland.svg
00:00.10 Ready
Osmo4 - surprisep
File View Play ?
Address D:\media\SVG\surprisep.svgz
00:00.08 Ready

Figure 5 – Example of a) static SVG content (Map_of_Iceland.svg, source: Wikipedia) and b) animated SVG content (surprise.svg, source: http://www.tinyline.com)

Table II and Table III indicate the statistics of the sequences that were used.

TABLE II
STATISTICS OF THE SVG STATIC SEQUENCES

Sequence name	File Size (kB)	Number of elements	Number of attributes	Number of points
2007-02-20_time_zones_white_bck.svg	2 397	4 707	13437	77 765
Africa_map_political-fr.svg	610	509	4518	23 458
America-blank-map-01.svg	650	889	2660	27 604
Centrales_Nucleaires_fr.svg	1 805	1 202	5649	61 109
cowboy.svg	437	2 735	2737	25 264
EspecesMammiferes Menacees_fr.svg	437	663	2623	41 760
GareNord1.001.svg	1 647	13 262	50890	158 085
gearflowers.svg	522	1 237	8782	9 374
Islam_by_country_01.svg	2 745	793	1362	133 103
Map_France_1477-fr.svg	1 251	1 435	15415	56 496
Map_of_Iceland.svg	830	4 398	11185	51 872
Map_of_the_Ancient_Rome_at_Caesar_time-fr.svg	1 231	406	4344	56 567
Mapa_Cor-de-Rosa.svg	2 830	3 476	21998	123 302
Mapa_municipal_del_domini_català.svg	3 459	2 632	25711	128864
Northern_Cities_Vowel_Shift.svg	298	246	1141	14 080
Paris_RER.svg	690	1 489	10211	31 516
plan.svg	38 395			
Pohjoisnapa.svg	1 437	2 804	6499	83 378
Quechuan_langs_map.svg	2 865	3 525	22676	123 762
svg-cards-2.0.svg	910	1 750	5740	78 390
tiger.svg	95	482	622	6 089
World_map_blank.svg	579	1 151	3697	41 083
Worldmap_wdb_combined.svg	855	9	30	47 237

TABLE III
STATISTICS OF THE SVG ANIMATED SEQUENCES

Sequence	File size	Number of elements	Number of animations
animate-elem-30-t.svg	21 860	74	17
animate-elem-37-t.svg	4 680	52	7
animate-elem-38-t.svg	7 126	9	42
animate-elem-80-t.svg	12 366	40	139
animate-elem-81-t.svg	6 322	8	68
animate-elem-82-t.svg	11 154	32	114
animate-elem-83-t.svg	8 710	19	90
bass2.svg	21 155	57	112
happybirthdayp.svg	32 241	63	110
retro4.svg	13 374	28	65
surprisep.svg	132 934	142	571
map.svg	21 860	74	17
cuisine.svg	21 898	82	5

C. *Results*

The first set of results concerns the memory consumption. Because it is difficult to efficiently measure memory usage of real-time applications on mobile devices and because few mobile players are freely available, we compare here the memory consumption of our player only with existing desktop players. These players are Adobe SVG Viewer 6.0, Firefox 2.0.0.8, Opera 9.25, Safari 3.0.3, and Renesis 0.7. We provide here the operating systems measurement. Even though it is not very precise, we will see that the results are already interesting. Additionally, since most players provide other functionalities than SVG visualization (i.e. browser features), in order to have a fair comparison, we give the difference between the memory usage when no content is loaded and when the content is loaded and displayed. In order to emulate the results on mobile devices, we modified the test sequences to set a width and height of the content to a small size (176x144).

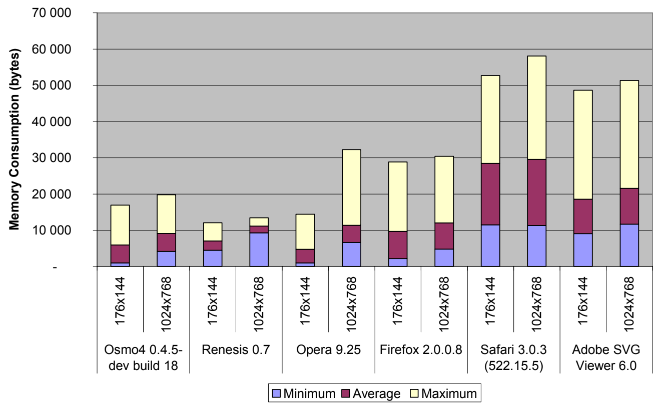

Figure 6 – Memory consumption for the sequences of Table II.

Figure 6 shows the minimum, average and maximum memory consumption for all tests and for two resolutions. It shows that our method for visualizing SVG content is efficient in terms of memory consumption. It performs better in both cases (small or big rendering size) than many existing desktop applications (Firefox, Safari, Internet Explorer with ASV6), with the additional advantage of being able to run on mobile devices. Only Opera and Renesis achieve comparable results. Opera performs better when the rendering size is small but consumes a higher amount of memory than GPAC when the rendering size is big, which could be a problem for Set Top Box implementations. Renesis also shows good results (for both rendering sizes). It is better than GPAC when the consumption is at its maximum but not on average nor on minimum.

Second, in order to give figures about the computational efficiency of our algorithms, we evaluated the average frame rate which can be achieved on different platforms. The test platforms are: average desktop PC (1.2 GHz Core Solo, 2 GB RAM), PDA Dell Axim X51v (624 MHz, 64 MB RAM), Smartphone SPV C 500 (200 MHz, 24 MB RAM).

These results are given in Table IV. The frame rate is an average of the frame rate computed every 30 frames. Additionally, we were not able to measure this number for other players either because they don't support animation (Firefox, Renesis) or for the others, because frame rate information is not provided.

For interested readers, all test sequences, detailed results and software implementation are available for download at http://www.enst.fr/~concolat/ToCE_data.zip.

TABLE IV
AVERAGE FRAME RATE ACHIEVABLE WITH THE GPAC PLAYER ON THE SVG ANIMATED SEQUENCES FROM TABLE III

Sequence	Average Frame Rate		
	PC	PDA	Phone
animate-elem-30-t.svg	270	45	25
animate-elem-37-t.svg	90	100	50
animate-elem-38-t.svg	70	60	30
animate-elem-80-t.svg	260	70	26
animate-elem-81-t.svg	220	150	80
animate-elem-82-t.svg	200	85	19
animate-elem-83-t.svg	110	30	15
bass2.svg	200	70	29
happybirthdayp.svg	170	33	16
retro4.svg	270	105	45
surprisep.svg	110	75	30
map.svg	120	35	20
cuisine.svg	330	45	25

We can see from this table that our proposed algorithms can achieve very good frame rates, even for complex animations. High frame rate (>100) can be reached on average desktop PC and reasonable frame rates (>25) can be reached on limited devices like our SPV mobile phone.

VI. CONCLUSION

We have described in this paper the general principles behind the visualization of multimedia scenes. We have then highlighted the problematic of the SVG visualization. Since this format is part of the selected formats for mobile devices, we have designed scene structures and according compositing and rendering algorithms which achieve interesting results both in terms of memory usage and animation frame rate, while remaining fully compliant with the standard. We believe these results are the consequence of innovative algorithms, in particular the SVG scene tree change-detection algorithm.

However, the proposed algorithms have some drawbacks. We can mention some problems with the order of definition and usage of inherited gradients and the fact that accessing the computed value requires a dedicated sub-tree traversal. These are items we will try to improve in future work.

REFERENCES

[1] "Scalable Vector Graphics (SVG) Tiny 1.2 Specification", W3C Candidate Recommendation 10 August 2006, http://www.w3.org/TR/SVGMobile12/
[2] "Synchronized Multimedia Integration Language (SMIL 2.1)", W3C Recommendation 13 December 2005, http://www.w3.org/TR/SMIL2/
[3] "Cascading Style Sheets, level 2 (CSS2) Specification", W3C Recommendation 12 May 1998, http://www.w3.org/TR/REC-CSS2/
[4] "Document Object Model (DOM) Level 2", W3C Recommendation 13 November 2000, http://www.w3.org/DOM/DOMTR
[5] "Information technology - Coding of audio-visual objects - Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF)", International Standard ISO/IEC 14496-20:2006
[6] M. Kostoulas,, M. Matsa., N. Mendelsohn., E. Perkins., A. Heifets., and M. Mercaldi., "XML screamer: an integrated approach to high performance XML parsing, validation and deserialization", Proc. of the 15th international Conference on World Wide Web, pp. 93-102. 2006.
[7] A. Cogliati, and P. Vuorimaa, "Optimized CSS Engine", Proc. of the 2nd International Conference on Web Information Systems and Technologies, pp. 206-213, 2006.
[8] J. Le Feuvre, C. Concolato, and J.-C. Moissinac, "GPAC: open source multimedia framework" Proc. of the 15th international Conference on Multimedia, pp. 1009-1012, 2007.
[9] "Over 250 Million Ikivo Powered Mobile SVG Devices Shipped", Press Release, available at http://www.ikivo.com/pdf/pressreleases/250_million_shipped.pdf
[10] "Dynamic and Interactive Multimedia Scenes", http://www.3gpp.org/specs/WorkItem-info/WI--34032.htm
[11] BitFlash SVG Tiny Player, http://www.bitflash.com/
[12] "Remote Events for XML (REX) 1.0", W3C Working Draft 13 October 2006, http://www.w3.org/TR/rex/
[13] "The XMLHttpRequest Object", W3C Working Draft 26 October 2007, http://www.w3.org/TR/XMLHttpRequest/
[14] "OpenVG - The Standard for Vector Graphics Acceleration", http://www.khronos.org/openvg/
[15] Jesse James Garret. "Ajax: A New Approach to Web Applications", http://adaptivepath.com/ideas/essays/archives/000385.php
[16] "Document Object Model (DOM) Level 2 Events Specification, Version 1.0", W3C Recommendation 13 November, 2000, http://www.w3.org/TR/DOM-Level-2-Events/
[17] "Efficient XML Interchange (EXI) Format 1.0", W3C Working Draft, 26 March 2008, http://www.w3.org/TR/2008/WD-exi-20080326/

Cyril Concolato received the Ingénieur (M.Sc.) degree in Telecommunications (2000) and the Ph.D. degree for his work on scene descriptions representations (2007) from the Ecole Nationale Supérieure des Télécommunications in Paris, France, where he is now Associate Professor in the Multimedia group of the Signal Processing and Image Department. His research interests are multimedia scene description languages, content adaptation and content distribution. He was an active contributor to the MPEG-4 BIFS standard and, in particular was editor of the Advanced Text and Graphics amendment. He is actively contributing and participating to the standardization activities of MPEG and W3C. He is one of the lead developers of the Open Source GPAC project.

Jean Le Feuvre received his Ingénieur (M.Sc.) degree in Telecommunications in 1999, from TELECOM Bretagne. Co-founder of Avipix, an MPEG-4 consulting and software provider company, he has been involved in MPEG standardisation since 2000. He joined TELECOM ParisTech in 2005 as Associate Professor within the Signal Processing and Image Department. Expert in multimedia delivery and rendering systems, he is the project leader and maintainer of GPAC, a rich media framework based on standard technologies (MPEG, W3C, and Web3D).

Jean-Claude Moissinac is Associate Professor within the Multimedia group of the Signal Processing and Image Department. His main research focus is defining tools to produce and consume multimedia contents within networks. He is also a member of the SVG working group of W3C. He is contributor to the SpipCarto and GPAC Open-Source projects.