SVG++ Documentation

Release 1.0

Oleg Maximenko

Nov 23, 2018
## Contents

1 About  
1.1 Installation ................................................. 2  
1.2 Tutorial ...................................................... 2  
1.3 Library Organization ....................................... 12  
1.4 Library Reference .......................................... 16  
1.5 Advanced Topics ........................................... 45  
1.6 FAQ .......................................................... 47  
1.7 Licensing ..................................................... 48  
1.8 Help and Feedback ......................................... 49  
1.9 Acknowledgements .......................................... 49
SVG++ library can be thought of as a framework, containing parsers for various SVG syntaxes, adapters that simplify handling of parsed data and a lot of other utilities and helpers for the most common tasks. Take a look at Tutorial to get the idea about what it is. Library sources available at https://github.com/svgpp/svgpp.

**SVG++ features**

- Is a header-only library
- Can be used with any XML parser
- Compile time configured - no virtual functions
- Minimal runtime overhead - you pay only for what you get
- Has wide range of possible uses:
  - Fully functional, conforming SVG viewers
  - Simple in-app SVG rasterizers
  - Import modules of vector editing software
  - Implementing path-only input of SVG format with minimal efforts in any graphics or math software
- Requires only Boost library itself (demo, tests and sample have their own requirements)
- Compatible with C++03, but requires conforming implementation, due to heavy usage of templates
- Supports any character type (char and wchar_t). Other (Unicode from C++11) were not tested, but should be ok.

**SVG 1.1 features covered by SVG++**

- Path data - parsing, simplification
- Transform list - parsing, simplification
- Color, including ICC color - parsing
- Lengths - highly configurable handling of SVG lengths, full support of all SVG length units
- Basic shapes - optional conversion to path
• Style attribute - parsing, taking in account differences in parsing presentation attributes and style properties
• Automatic marker positions calculation
• Viewport and viewbox handling

SVG++ is NOT
• An SVG viewer or library that produces raster image from SVG (though C++ demo app included, which renders SVG with AGG, GDI+ or Skia graphics library)
• Renders anything itself

Contents

1.1 Installation

The latest version of SVG++ can be downloaded from SVG++ repository, at https://github.com/svgpp/svgpp, or as a package at https://github.com/svgpp/svgpp/archive/master.zip

SVG++ requires Boost library (was tested with Boost >= 1.55). Only header-only libraries from Boost are used, no build required.

SVG++ consists of header files only and does not require any libraries to link against. Including svgpp/svgpp.hpp will be sufficient for most projects.

Library was tested with this compilers:
• GCC 4.8 (in C++11 mode it requires Boost >= 1.56)
• Clang 3.2

Note: Compilation of module that uses a lot of SVG++ features may require large amount of RAM. Please be aware of this when compiling on VMs.

1.2 Tutorial

This tutorial introduces SVG++ features by adding them one-by-one into some sample application.

All SVG++ headers may be included through this one:

```cpp
#include <svgpp/svgpp.hpp>
```

All SVG++ code is placed in svgpp namespace. We’ll import the entire namespace in our sample.

1.2.1 Handling Shapes Geometry

Basic pattern of SVG++ usage: call document_traversal::load_document method, passing XML element and context to it. Library will call methods of context, passing the parsed data.
#include <svgpp/svgpp.hpp>

using namespace svgpp;

class Context
{
public:
    void path_move_to(double x, double y, tag::coordinate::absolute);
    void path_line_to(double x, double y, tag::coordinate::absolute);
    void path_cubic_bezier_to(
        double x1, double y1,
        double x2, double y2,
        double x, double y,
        tag::coordinate::absolute);
    void path_quadratic_bezier_to(
        double x1, double y1,
        double x, double y,
        tag::coordinate::absolute);
    void path_elliptical_arc_to(
        double rx, double ry,
        double x_axis_rotation,
        bool large_arc_flag,
        bool sweep_flag,
        double x, double y,
        tag::coordinate::absolute);
    void path_close_subpath();
    void path_exit();
    void on_enter_element(tag::element::any);
    void on_exit_element();
};

typedef
boost::mpl::set<
    // SVG Structural Elements
    tag::element::svg,
    tag::element::g,
    // SVG Shape Elements
    tag::element::circle,
    tag::element::ellipse,
    tag::element::line,
    tag::element::path,
    tag::element::polygon,
    tag::element::polyline,
    tag::element::rect
>::type processed_elements_t;

void loadSvg(xml_element_t xml_root_element)
{
    Context context;
    document_traversal<
        processed_elements<processed_elements_t>,
        processed_attributes<traits::shapes_attributes_by_element>
    >::load_document(xml_root_element, context);
}

#document_traversal is a facade that provides access to most library capabilities.
In most cases only some subset of SVG elements is needed, so we pass named template parameter processed_elements
to document_traversal template class. In our case it is processed_elements_t - boost::mpl::set
that combines traits::shape_elements (enumerates SVG shapes) with svg and g elements.

SVG++ references SVG element types by tags.

We choose SVG attributes subset and pass it as processed_attributes parameter. traits::shapes_attributes_by_element contains attributes, that describe geometry of all shapes ({x, y, width, height, rx and ry} for rect, {d} for path etc.).

In this sample the same context instance is used for all SVG elements. Context::on_enter_element(element_tag) is called when moving to child SVG element, type of child element passed as tag in the only argument (tag::element::any is a base class for all element tags). on_exit_element() is called when processing of child element is finished:

<table>
<thead>
<tr>
<th>XML element</th>
<th>Call to context</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;svg&gt;</td>
<td>on_enter_element(tag::element::svg())</td>
</tr>
<tr>
<td>&lt;rect x=&quot;100&quot; y=&quot;200&quot; /&gt;</td>
<td>on_exit_element()</td>
</tr>
<tr>
<td>&lt;g&gt;</td>
<td>on_enter_element(tag::element::g())</td>
</tr>
<tr>
<td>&lt;rect x=&quot;300&quot; y=&quot;100&quot; /&gt;</td>
<td>on_exit_element()</td>
</tr>
<tr>
<td>&lt;/g&gt;</td>
<td>on_exit_element()</td>
</tr>
<tr>
<td>&lt;/svg&gt;</td>
<td>on_exit_element()</td>
</tr>
</tbody>
</table>

Calls like path_XXXX except path_exit correspond to SVG path data commands. path_exit is called after path data attribute was parsed.

SVG++ by default (see Path Policy for details):

- converts relative coordinates to absolute ones;
- commands for horizontal and vertical lines (H, h, V, v) converts to calls to path_line_to with two coordinates;
- shorthand/smooth curveto and shorthand/smooth quadratic Bézier curveto replaces with calls with full parameters list.

SVG++ by default converts basic shapes to path (see Basic Shapes Policy for details).

### 1.2.2 XML Parser

We didn’t declared xml_element_t yet. Let’s use RapidXML NS library (it is a clone of RapidXML with namespace handling added) that comes with SVG++ in the third_party/rapidxml_ns/rapidxml_ns.hpp file. It’s a single header library, so we just need to point to its header:

```cpp
#include <rapidxml_ns/rapidxml_ns.hpp>
```

Then we must include SVG++ policy for chosen XML parser:

```cpp
#include <svgpp/policy/xml/rapidxml_ns.hpp>
```

XML policy headers don’t include parser header because their location and names may differ. The programmer must include appropriate XML parser header herself before including policy header.

Setting appropriate XML element type for RapidXML NS parser:
You can find the full cpp file here: src/samples/sample01a.cpp.

### 1.2.3 Handling Transformations

Just add `tag::attribute::transform` to `processed_attributes` list and `transform_matrix` method to `Context` class:

```cpp
void transform_matrix(const boost::array<double, 6> & matrix);
```  
```cpp
typedef
  boost::mpl::insert<
    traits::shapes_attributes_by_element,
    tag::attribute::transform
  >::type processed_attributes_t;
/* ... */
```  
```cpp
document_traversal<
  processed_elements<processed_elements_t>,
  processed_attributes<processed_attributes_t>
>::load_document(xml_root_element, context);
```  
Passed matrix array `[a b c d e f]` correspond to this matrix:

```
[  0  1  0  0  0  0 ]
[ b c e d f a ]
```

The default SVG++ behavior is to join all transformations in `transform` attribute into single affine transformation matrix.

Source file: src/samples/sample01b.cpp.

### 1.2.4 Handling Viewports

The `svg` element may be used inside SVG document to establish a new viewport. To process new viewport coordinate system and new user coordinate system several attributes must be processed (`x`, `y`, `width`, `height`, `preserveAspectRatio`, `viewbox`). SVG++ will do it itself if we set `policy::viewport::as_transform Viewport Policy`

```cpp
document_traversal<
  processed_elements<processed_elements_t>,
  processed_attributes<processed_attributes_t>,
  viewport_policy<policy::viewport::as_transform>
>::load_document(xml_root_element, context);
```  
we also must append viewport attributes to the list of processed attributes:
Please note that this cryptic code just merges predefined sequences `traits::shapes_attributes_by_element` and `traits::viewport_attributes` with `tag::attribute::transform` attribute into single MPL sequence equivalent to the following:

```cpp
typedef boost::mpl::set<
    // Transform attribute
    tag::attribute::transform,
    // Viewport attributes
    tag::attribute::x,
    tag::attribute::y,
    tag::attribute::width,
    tag::attribute::height,
    tag::attribute::viewBox,
    tag::attribute::preserveAspectRatio,
    // Shape attributes for each shape element
    boost::mpl::pair<tag::element::path, tag::attribute::d>,
    boost::mpl::pair<tag::element::rect, tag::attribute::x>,
    boost::mpl::pair<tag::element::rect, tag::attribute::y>,
    boost::mpl::pair<tag::element::rect, tag::attribute::width>,
    boost::mpl::pair<tag::element::rect, tag::attribute::height>,
    boost::mpl::pair<tag::element::circle, tag::attribute::cx>,
    boost::mpl::pair<tag::element::circle, tag::attribute::cy>,
    boost::mpl::pair<tag::element::circle, tag::attribute::r>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::cx>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::cy>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::rx>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::ry>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::ry>,
    boost::mpl::pair<tag::element::ellipse, tag::attribute::ry>,
    boost::mpl::pair<tag::element::line, tag::attribute::x1>,
    boost::mpl::pair<tag::element::line, tag::attribute::y1>,
    boost::mpl::pair<tag::element::line, tag::attribute::x2>,
    boost::mpl::pair<tag::element::line, tag::attribute::y2>,
    boost::mpl::pair<tag::element::polyline, tag::attribute::points>,
    boost::mpl::pair<tag::element::polygon, tag::attribute::points>
> ::type processed_attributes_t;
```

Now SVG++ will call the existing method `transform_matrix` to set new user coordinate system. And we must add some methods that will be passed with information about new viewport:

```cpp
void set_viewport(double viewport_x, double viewport_y, double viewport_width, double viewport_height);
void set_viewbox_size(double viewbox_width, double viewbox_height);
```

(continues on next page)
The full cpp file for this step can be found here: src/samples/sample01c.cpp.

### 1.2.5 Creating Contexts

Until now only one instance of context object was used for entire SVG document tree. It is convenient to create context instance on stack for each SVG element processed. This behavior is controlled by context factories, passed by context_factories parameter of document_traversal template class.

Context factories is a Metafunction Class that receives parent context type and child element tag as parameters and returns context factory type.

This sample application processes structural elements (svg and g) and shape elements (path, rect, circle etc). For the structural elements only transform attribute is processed, and for the shape elements - transform and attributes describing shape. So we can divide Context context class forBaseContext and ShapeContext subclass:

```cpp
class BaseContext {
public:
    void on_exit_element();
    void transform_matrix(const boost::array<double, 6> & matrix);
    void set_viewport(double viewport_x, double viewport_y, double viewport_width, double viewport_height);
    void set_viewbox_size(double viewport_width, double viewport_height);
    void disable_rendering();
};

class ShapeContext : public BaseContext {
public:
    ShapeContext(BaseContext const & parent);
    void path_move_to(double x, double y, tag::coordinate::absolute);
    /* ... other path methods ... */
};

struct ChildContextFactories {
    template<class ParentContext, class ElementTag, class Enable = void>
    struct apply {
        // Default definition handles "svg" and "g" elements
        typedef factory::context::on_stack<BaseContext> type;
    };
};

// This specialization handles all shape elements (elements from traits::shape_elements sequence)
template<class ElementTag>
struct ChildContextFactories::apply<BaseContext, ElementTag, typename boost::enable_if<boost::mpl::has_key<traits::shape_elements, ElementTag>> ::type> {
    typedef factory::context::on_stack<ShapeContext> type;
};
```
factory::context::on_stack<ChildContext> factory creates context object ChildContext, passing reference to parent context in ChildContext constructor. Lifetime of context object - until processing of element content (child elements and text nodes) is finished. on_exit_element() is called right before object destruction.

ChildContextFactories is passed to document_traversal:

```cpp
document_traversal<
  /* ... */
  context_factories<ChildContextFactories>
> :: load_document(xml_root_element, context);
```

Source file: src/samples/sample01d.cpp.

### 1.2.6 The use Element Support

The use element is used to include/draw other SVG element. If use references svg or symbol, then new viewport and user coordinate system are established.

To add support for use in our sample we:

- Add tag::element::use to the list of processed elements, and tag::attribute::xlink::href to the list of processed attributes (x, y, width and height already included through traits::viewport_attributes).
- Create context class UseContext to be used for use element, that will collect x, y, width, height and xlink:href attributes values.
- After processing all use element attributes (in method UseContext::on_exit_element()), look inside document for element with given id and load it with call to document_traversal_t::load_referenced_element<...>::load().
- Implement Viewport Policy requirement - svg and symbol context must have method:

  ```cpp
  void get_reference_viewport_size(double & width, double & height);
  ```

  that returns size of the viewport set in referenced use element. One of possible variant is creation of new context ReferencedSymbolOrSvgContext.

Full implementation is in file: src/samples/sample01e.cpp.

### 1.2.7 Calculating Marker Positions

SVG++ may solve complex task of calculating orientations of markers with attribute orient="auto". Let’s set Markers Policy that enables this option:

```cpp
document_traversal<
  /* ... */
  markers_policy<policy::markers::calculate_always>
> /* ... */
```

Then add Marker Events method to ShapeContext:

```cpp
void marker(marker_vertex v, double x, double y, double directionality, unsigned_˓
  marker_index);
```

The sample (src/samples/sample01f.cpp) just shows how to get marker positions. To implement full marker support we also need to process marker, marker-start, marker-mid and marker-end properties and process marker element (similar to processing of use element). Demo application may give some idea about this.
### 1.2.8 Processing of stroke and stroke-width Properties

Adding **stroke-width** property processing is trivial - just add `tag::attribute::stroke_width` to the list of processed attributes, and add method, that receives value, to the context class:

```cpp
void set(tag::attribute::stroke_width, double val);
```

Property **stroke** has complex type `<paint>`:

```cpp
```

that is why so many methods are required to receive all possible values of the property:

```cpp
void set(tag::attribute::stroke_width, double val);
void set(tag::attribute::stroke, tag::value::none);
void set(tag::attribute::stroke, tag::value::currentColor);
void set(tag::attribute::stroke, color_t color, tag::skip_icc_color = tag::skip_icc_color());
template<class IRI>
void set(tag::attribute::stroke tag, IRI const & iri);
template<class IRI>
void set(tag::attribute::stroke tag, tag::iri_fragment, IRI const & fragment);
template<class IRI>
void set(tag::attribute::stroke tag, IRI const & tag::value::none val);
template<class IRI>
void set(tag::attribute::stroke tag, tag::iri_fragment, IRI const & fragment, tag::value::none val);
template<class IRI>
void set(tag::attribute::stroke tag, IRI const & tag::value::currentColor val);
template<class IRI>
void set(tag::attribute::stroke tag, tag::iri_fragment, IRI const & fragment, tag::value::currentColor val);
template<class IRI>
void set(tag::attribute::stroke tag, IRI const & color_t val, tag::skip_icc_color = tag::skip_icc_color());
template<class IRI>
void set(tag::attribute::stroke tag, tag::iri_fragment, IRI const & fragment, color_t val, tag::skip_icc_color = tag::skip_icc_color());
```

Default **IRI Policy** is used that distinguishes absolute IRIs and local IRI references to fragments in same SVG document.

Source code: `src/samples/sample01g.cpp`.

**Note:** `svgpp_parser_impl.cpp` file was added to the project and a couple of macros was added at the start of the `sample01g.cpp` to get around Visual C++ 2015 “compiler out of memory” problem. See description of this solution.
1.2.9 Custom Color Factory

Suppose that default SVG++ color presentation as 8 bit per channel RGB value packed in int doesn’t suit our needs. We prefer to use some custom type, e.g. boost::tuple (same as C++11 std::tuple):

```cpp
typedef boost::tuple<unsigned char, unsigned char, unsigned char> color_t;
```

In this case we need our own Color Factory, that creates our custom color from components values, that was read from SVG:

```cpp
struct ColorFactoryBase
{
    typedef color_t color_type;
    static color_type create(unsigned char r, unsigned char g, unsigned char b)
    {
        return color_t(r, g, b);
    }
};
```

```cpp
typedef factory::color::percentage_adapter<ColorFactoryBase> ColorFactory;
document_traversal<
    /* ... */
    color_factory<ColorFactory>
> /* ... */;
```

Usage of factory::color::percentage_adapter frees us from implementing create_from_percent method in our Color Factory.

Source file: src/samples/sample01h.cpp.

1.2.10 Correct Length Handling

On next step (src/samples/sample01i.cpp) of sample evolution we will add correct handling of length, that takes in account device resolution (dpi) and changes of viewport size by svg and symbol elements, that affects lengths, which are set in percent. So we:

- Add toBaseContext class constructor that receives device resolution in dpi (this constructor is only called by ourselves from loadSvg function).
- Add length_factory_ field and access function. length_factory_ settings (resolution, viewport size) will be passed to child contexts in copy constructor.
- In BaseContext::set_viewport and BaseContext::set_viewbox_size methods pass viewport size to the Length Factory.
- Set Length Policy, that will ask BaseContext class for Length Factory instance:

```cpp
document_traversal<
    /* ... */
    length_policy<policy::length::forward_to_method<BaseContext> >
> /* ... */;
```

```cpp
classBaseContext: public StylableContext
{
    public:
```
BaseContext(double resolutionDPI)
{
    length_factory_.set_absolute_units_coefficient(resolutionDPI, tag::length_˓→units::in());
}

/* ... */

// Viewport Events Policy
void set_viewport(double viewport_x, double viewport_y, double viewport_width,´˓→double viewport_height)
{
    length_factory_.set_viewport_size(viewport_width, viewport_height);
}

void set_viewbox_size(double viewbox_width, double viewbox_height)
{
    length_factory_.set_viewport_size(viewbox_width, viewbox_height);
}

// Length Policy interface
typedef factory::length::unitless<> length_factory_type;

length_factory_type const & length_factory() const
{
    return length_factory_;}

private:
    length_factory_type length_factory_;}

According to SVG Specification, the size of the new viewport affects attributes of element that establish new view-
port (except x, y, width and height attributes). As our Length Factory converts lengths to numbers immediately,
we need to pass new viewport size to Length Factory before processing other attributes. To do so we will use
get_priority_attributes_by_element parameter of Attribute Traversal Policy:

struct AttributeTraversal: policy::attribute_traversal::default_policy
{
    typedef boost::mpl::if_< ˓→// If element is 'svg' or 'symbol'...
        boost::mpl::has_key<
            boost::mpl::set<
                tag::element::svg,
                tag::element::symbol
            >,
            boost::mpl::l1
        >,
        boost::mpl::vector< ˓→// ... load viewport-related attributes first ...
            tag::attribute::x,
            tag::attribute::y,
            tag::attribute::width,
            tag::attribute::height,
            tag::attribute::viewBox,
            tag::attribute::preserveAspectRatio,
        >, ˓→viewbox size
;
(continues on next page)
notify_context<tag::event::after_viewport_attributes>::type,
boost::mpl::empty_sequence
> get_priority_attributes_by_element;
};
document_traversal<
/* ... */
attribute_traversal_policy<AttributeTraversal>
> /* ... */;

Now we are sure that `BaseContext::set_viewport` (and `BaseContext::set_viewbox_size`) will be called before other attributes are processed.

### 1.2.11 Text Handling

On the next step (src/samples/sample01j.cpp) we will implement basic handling of `text` elements:

- Create new `TextContext` child of `BaseContext` that will receive `text` element data. `set_text` method will be called by SVG++ to pass character data content of element:

```cpp
template<class Range>
void set_text(Range const & text) {
    text_content_.append(boost::begin(text), boost::end(text));
}
```

- Add specialization of `ChildContextFactories` class that will create `TextContext` class for `text` elements (`tag::element::text`).
- Add `tag::element::text` to the list of processed elements `processed_elements_t`.

### 1.3 Library Organization

#### 1.3.1 Context

Main pattern of SVG++ usage: programmer calls some library function, passing reference to context and reference to XML element or value of XML attribute; library calls static methods of corresponding Events Policy, passing context and parsed values as arguments.

*Events Policies* (Value Events Policy, Transform Events Policy, Path Events Policy etc) know how to pass value to corresponding context type. Default *Events Policies* treats context as an object, calling its methods.

#### 1.3.2 Components

Below are shown the main SVG++ components, starting from the lowest level. Every following is based on the preceding:

- **Value Parsers** *Value Parsers* are functions that convert string values of attributes and CSS properties to calls of user functions with convenient presentation of SVG data. Some of them use *grammars.*
E.g., attribute value `x="16mm"` may be passed as corresponding `double` value, taking into account units, and `d="M10 10 L15 100"` may become sequence of calls like `path_move_to(10,10); path_line_to(15, 100);`.

**Adapters** Value Parsers as much as possible saves SVG information, this allows, for example, to use SVG++ for SVG DOM generation. In other applications this information may be excessive. SVG++ provides some adapters, that may simplify processing of SVG. Adapters are configured by policies.

**Attribute Dispatcher** Attribute Dispatcher object is created internally for each element. It chooses and calls corresponding Value Parser for each attribute. Besides that Attribute Dispatcher manages adapters that processed several attributes of same element. For example, adapter that converts `line` element to `path` must collect values of `x1`, `y1`, `x2` and `y2` attributes - this is managed by Attribute Dispatcher.

**Attribute Traversal**
- Attribute Traversal object is created internally for each element and it calls methods of Attribute Dispatcher.
- Finds out numeric id of attribute by its name.
- Parses style attribute, so that values of attributes and values of CSS properties are handled identically.
- Checks presence of mandatory attributes in element.
- “Hides” presentation attribute, if this property is already set in the style attribute.
- Allows attribute ordering.

**Document Traversal**
- Traverses SVG document tree, processes elements selected for processing by the programmer.
- Checks content model, i.e. whether each child element is allowed for this parent.
- Creates instances of Attribute Dispatcher and Attribute Traversal and passes processing to them for each element.
- Passes to the user code child text nodes of SVG elements that are allowed to carry text content by SVG Specification.

Document Traversal provides convenient access to entire library and, in most cases, it is the only SVG++ component with which programmer interacts.

**Grammars** are quite independent components and can be easily used separately.

Value Parsers have simple interface and can be easily called from application if for some reason traversing of SVG document tree by Document Traversal isn’t applicable or only few attributes need to be parsed.

Attribute Traversal and Attribute Dispatcher aren’t described in the documentation and unlikely to be used from outside.

**1.3.3 Tags**

SVG++ widely uses tag concept to reference various SVG entities in compile time with overload resolution and metaprogramming techniques. Tag here is just an empty class.

```cpp
namespace tag {
    namespace element {
        (continues on next page)
    }
}
```

1.3. Library Organization 13
Each SVG element corresponds to tag in tag::element C++ namespace, and each SVG attribute (or property) corresponds to tag in tag::attribute namespace. Attributes from XML namespace xlink corresponds to tags in namespace tag::attribute::xlink, and attributes from XML namespace xml corresponds to tags in namespace tag::attribute::xml. There are also other tags, described in other parts of the documentation.

1.3.4 Named Class Template Parameters

SVG++ widely uses named class template parameters for compile-time library configuration. It looks like this:

```cpp
svgpp::document_traversal<
    svgpp::length_policy<SomeUserLengthPolicy>,
    svgpp::path_policy<SomeUserPathPolicy>
/* ... */
>::load_document(/* ... */);
```

In this example SomeUserLengthPolicy type is passed as length_policy parameter, and SomeUserPathPolicy type is passed as path_policy parameter.

Named class template parameters are passed through SVG++ components down to Value Parsers level.

1.3.5 Library Customization

Policies allows customization of most library aspects. There are two ways of setting policy:

1. Pass policy as a named class template parameter. For example:

```cpp
document_traversal<
    length_policy<UserLengthPolicy>
>::load_document(/* ... */);
```

2. Create specialization of class default_policy for the context type in proper C++ namespace:
namespace svgpp { namespace policy { namespace length {
    template<>
    struct default_policy<UserContext>: UserLengthPolicy
    {};
}}}

1.3.6 XML Parser

SVG++ uses external XML parsing libraries. Interaction with XML parser is handled by specialization of XML Policy classes.

XMLElement template parameter is used to automatically choose XML Policy for XML parser used.

Programmer must include XML parser library header files, after that include header file of corresponding XML Policy from SVG++ and only after that include other SVG++ headers. For example:

```cpp
#include <rapidxml_ns/rapidxml_ns.hpp>
#include <svgpp/policy/xml/rapidxml_ns.hpp>
#include <svgpp/svgpp.hpp>
```

Below are XML parsing libraries supported by SVG++, their respective XML Policy header files and expected XML::Element type:

<table>
<thead>
<tr>
<th>XML Parser Library</th>
<th>Policy header</th>
<th>XMLElement template parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>RapidXML NS</td>
<td>&lt;svgpp/policy/xml/rapidxml_ns.hpp&gt;</td>
<td>rapidxml_ns::xml_node&lt;Ch&gt; const *</td>
</tr>
<tr>
<td>libxml2</td>
<td>&lt;svgpp/policy/xml/libxml2.hpp&gt;</td>
<td>xmlNode *</td>
</tr>
<tr>
<td>MSXML</td>
<td>&lt;svgpp/policy/xml/msxml.hpp&gt;</td>
<td>IXMLDOMElement *</td>
</tr>
<tr>
<td>Xerces</td>
<td>&lt;svgpp/policy/xml/xerces.hpp&gt;</td>
<td>xercesc::DOMEElement const *</td>
</tr>
</tbody>
</table>

1.3.7 Strings

SVG++ supports different character types - char and wchar_t, and on supporting compilers char16_t and char32_t. Character type is defined by XML parsing library used.

Strings are passed to the user code by some unspecified model of Forward Range concept. Example of processing:

```cpp
struct Context {
    template<class Range>
    void set(svgpp::tag::attribute::result, Range const & r) {
        std::string value;
        value_.assign(boost::begin(r), boost::end(r));
    }
};
```

If template function can’t be used (e.g. it is virtual function), then boost::any_range can be used as string range type.

1.3.8 CSS Support

SVG++ parses properties in style attribute, if style processing is enabled by the programmer.
SVG++ doesn’t support CSS cascading and CSS stylesheet in style element. It may be handled, if needed, by some other component, that provides result as style attribute.

1.4 Library Reference

1.4.1 Document Traversal

document_traversal Class

document_traversal template class is a facade that provides access to most features of SVG++ library.
document_traversal contains only static methods. Each document_traversal method receives context and XML element parameters. Context is a user defined object that will receive parsed data. XML element has type of XML element in chosen XML parser.

Named template parameters of document_traversal class allows to configure almost each aspect of SVG processing.

```cpp
template<class Args...>
struct document_traversal
{
    template<class XMLElement, class Context>
    static bool load_document(XMLElement const & xml_root_element, Context & context)
    {
        return load_expected_element(xml_root_element, context, tag::element::svg());
    }

    template<class XMLElement, class Context, class ElementTag>
    static bool load_expected_element(
        XMLElement const & xml_element,
        Context & context,
        ElementTag expected_element);

    template<class RefArgs...>
    struct load_referenced_element
    {
        template<class XMLElement, class Context>
        static bool load(XMLElement const & xml_element, Context & parent_context);
    }
};
```

Methods

```cpp
template<class XMLElement, class Context>
static bool load_document(XMLElement const & xml_root_element, Context & context);

load_document - is a shortcut for load_expected_element that receives root element (svg) of SVG document as xml_root_element parameter.

template<class XMLElement, class Context, class ElementTag>
static bool load_expected_element(
    XMLElement const & xml_element,
    Context & context,
    ElementTag expected_element);```

load_expected_element loads xml_element and its descendands and passes parsed data to context.

load_expected_element expects that XML name of xml_element corresponds to ElementTag. Otherwise exception unexpected_element_error is thrown (see Error Policy).

\[
\begin{align*}
\text{template<class RefArgs...>}& \\
\text{template<class XMLElement, class Context>}& \\
\text{static bool load_referred_element<RefArgs...>::load(}& \\
\text{XMLElement const & xml_element,}& \\
\text{Context & parent_context);}& \\
\end{align*}
\]

load_referred_element mainly used to load SVG elements that are referenced by other SVG element, e.g. use element or gradients. Unlike load_expected_element, allowed XML name of xml_element isn’t limited to one and is passed as expected_elements sequence.

Named class template parameters of document_traversal::load_referred_element

- **expected_elements (required)** Value is Associative Sequence. Contains list of tags of expected elements for xml_element. If XML name of xml_element doesn’t correspond to any from the list, exception unexpected_element_error (see Error Policy) is thrown.

  Note: traits::reusable_elements contains list of elements, that can be referenced by use element.

- **referencing_element (optional)** Value is tag of element that references xml_element. It is used only if Viewport Policy requires processing of viewport by SVG++ (for correct processing of svg and symbol elements, referenced by image or use elements).

- **processed_elements or ignored_elements (optional)** Only one may be set. See description of document_traversal parameters with the same names. Allows to over-ride document_traversal settings processed_elements/ignored_elements for the passed SVG element. Child elements will be processed with document_traversal settings.

**document_traversal Named Class Template Parameters**

- **ignored_elements and processed_elements** One of them must be set to limit SVG elements processed. It must be model of Associative Sequence (e.g. boost::mpl::set), containing element tags.

  If processed_elements is set, then only the listed elements are processed, otherwise if ignored_elements is set, then only non-listed elements are processed.

- **ignored_attributes and processed_attributes** One of them must be set to limit SVG attributes processed. It must be model of Associative Sequence, containing attribute tags. Also it may contain pairs <element tag, attribute tag> like this boost::mpl::pair<tag::element::g, tag::attribute::transform>, such pair is matched if both processed element and processed attribute matches tags.

  If processed_attributes is set, then only listed attributes are processed, otherwise if ignored_attributes is set, then only non-listed attributes are processed.

- **passthrough_attributes (optional)** Is a Associative Sequence, that contains attribute tags. Values of listed attributes aren’t parsed by SVG++, and passed to the user code as string.

- **context_factories (optional)** See Context Factories.
attribute_traversal_policy (optional) See Attribute Traversal Policy.

Context Factories

When document_traversal traverses SVG tree, it, for each SVG element met, chooses context to be used to pass attribute values and character data by corresponding event policy. To configure this behavior context_factories parameter is used.

context_factories parameter accepts Metafunction Class, that receives:

- ParentContext - parent context type (context used for parent SVG element);
- ElementTag - element tag (corresponds to the SVG element met),

and returns Context Factory.

typedef
typename context_factories::template apply<ParentContext, ElementTag>::type
selected_context_factory;

SVG++ provides several Context Factories:

template<class ParentContext, class ElementTag>
class factory::context::same;

The new context object isn’t created, parent context will be used.
on_enter_element(ElementTag()) and on_exit_element() methods of parent context will be called at the start and at the end of element processing respectively.

template<class ChildContext>
class factory::context::on_stack;

The new object of type ChildContext is created on stack. Constructor of the object is passed with the reference to the parent context. After element processing is finished, method ChildContext::on_exit_element() is called before destruction. Lifetime of context object matches processing of SVG element content (element attributes, child elements and text nodes).

factory::context::on_stack_with_xml_element is the same as factory::context::on_stack, but ChildContext constructor receives second parameter - XML element. Its type depends on XML parser used.

template<
class ElementTag,
class ChildContextPtr,
class ChildContext = typename boost::pointee<ChildContextPtr>::type
>
class get_ptr_from_parent;

Pointer ChildContextPtr to context object is requested from parent context by call to the method get_child_context(ElementTag()). Pointer can be raw pointer or some smart pointer. ChildContext::on_exit_element() is called after element processing is finished.

Attribute Traversal Policy

Attribute Traversal Policy configures order and other aspects of SVG attributes and CSS properties processing.
struct attribute_traversal_policy_concept
{
    static const bool parse_style = /* true or false */;
    static const bool css_hides_presentation_attribute = /* true or false */;

typedef /* Metafunction class */ get_priority_attributes_by_element;
.typedef /* Metafunction class */ get_deferred_attributes_by_element;
.typedef /* Metafunction class */ get_required_attributes_by_element;
};

parse_style = true Contents of style attribute is parsed as a sequence of semicolon-separated pairs property-value.

css_hides_presentation_attribute = true Is checked only if parse_style = true.
    If the same property is set both in style attribute and as a presentation attribute, then only value in style attribute will be parsed (it has higher priority according to SVG).
    If css_hides_presentation_attribute = false, then memory usage is lower, but both values (from style attribute and a presentation attribute) of the same property will be parsed and passed to the user code in the arbitrary order.

get_priority_attributes_by_element, get_deferred_attributes_by_element and get_required_attributes_by_element
    They are Metafunction classes, receiving element tag and returning Forward Sequence of attribute tags.
    Attributes, returned by get_priority_attributes_by_element metafunction for current element, will be processed before all others in the same order as in sequence.
    Attributes, returned by get_deferred_attributes_by_element metafunction for current element, will be processed after all others in the same order as in sequence.

    Note: Sequences, returned by get_priority_attributes_by_element and get_deferred_attributes_by_element, besides attribute tags may contain elements like notify_context<EventTag>. EventTag is an arbitrary tag, that will be passed to notify method of the context. notify(EventTag()) will be called right after all previous attributes in the sequence are processed.

If element lacks any attribute from the sequence, returned by get_required_attributes_by_element metafunction for this element, then required_attribute_not_found_error exception will be thrown (see Error Policy). SVG++ contains definition of traits::element_required_attributes metafunction that returns mandatory attributes for element, according to SVG Specification. It can be used like this:

struct my_attribute_traversal_policy
{
    /* ... other members ... */
    typedef boost::mpl::quote1<traits::element_required_attributes>
        get_required_attributes_by_element;
};

1.4.2 How parsed values are passed to context

To find out how value of the SVG attribute will be passed to context, following algorithm should be applied:

1. If attribute tag is included in passthrough_attributes sequence, then its value will be passed by Value Events Policy as a string.
2. If for this element and attribute traits::attribute_type<ElementTag, AttributeTag>::type is tag::type::string, then value will also be passed by Value Events Policy as a string.

3. Otherwise type of the attribute should be found in its description in SVG Specification.

4. Attribute values of type <path data> (e.g. d attribute of path element) are described in Path section.

5. Attribute values of type <transform list> (e.g. transform attribute) are described in Transform section.

6. All others are passed through Value Events Policy.

Value Events Policy

Value Events Policy Concept

Value Events Policy is a class, containing static set methods, that receives reference to context object as a first argument, attribute tag as a second and a source tag as a third. Source tag is one of two types: tag::source::css and tag::source::attribute (they have common base tag::source::any). Source tag shows where value comes from – CSS value in style attribute or from separate SVG/XML attribute. Number and types of other parameters depends on an attribute type.

```cpp
struct value_events_policy
{
    static void set(Context & context, AttributeTag tag, SourceTag source, ValueType1 const & value1);
    static void set(Context & context, AttributeTag tag, SourceTag source, ValueType1 const & value1, ValueType2 const & value2);
    /*...*/
};
```

class::value_events::forward_to_method used by default forward calls to set methods of context object:

```cpp
template<class Context>
struct forward_to_method
{
    template<class AttributeTag, class ...Args>
    static void set(Context & context, AttributeTag tag, tag::source::any const &, Args. && args)
    {
        context.set(tag, args...);
    }
};
```

Note: Source tag is dropped by default Value Events Policy, because default Attribute Traversal Policy processes only one value of the same property, even if both are provided (css_hides_presentation_attribute = true).

Note: Default Value Events Policy doesn’t pass tag::value::inherit values of properties and presentation attributes, that are inherited (see policy::value_events::skip_inherit). inherit value for such attributes is equivalent to its absence.

Example of default Value Events Policy usage:
#include <svgpp/svgpp.hpp>

using namespace svgpp;

struct Context
{
    void set(tag::attribute::amplitude, double value)
    {
        std::cout << value << "\n";
    }
};

void func()
{
    Context context;
    value_parser<tag::type::number>::parse(tag::attribute::amplitude(), context,
    std::string("3.14"), tag::source::attribute());
}

Example of own Value Events Policy implementation. Created policy::value_events::default_policy template class specialization for our context type (let it be boost::optional<double> in our example):

namespace svgpp { namespace policy { namespace value_events {

    template<>
    struct default_policy<boost::optional<double> >
    {
        template<class AttributeTag>
        static void set(boost::optional<double> & context, AttributeTag tag,
            tag::source::any, double value)
        {
            context = value;
        }
    };

} } }

void func()
{
    boost::optional<double> context;
    value_parser<tag::type::number>::parse(tag::attribute::amplitude(), context,
    std::string("3.14"), tag::source::attribute());
    if (context)
        std::cout << *context << "\n";
}

Literal Values  If literal is one of attribute possible values, then this value will cause call with tag from tag::value namespace. Example of attributes that may have literal values:

gradientUnits = "userSpaceOnUse | objectBoundingBox"
clip-path = "<funciri> | none | inherit"

gradientUnits is limited to two literal values. clip-path, besides none and inherit literal values may get values of <FuncIRI> type.

Example of context implementation, that receives values of gradientUnits attributes:

class GradientContext

(continues on next page)
GradientContext()
    : m_gradientUnitsUserSpace(false)
{}

void set(tag::attribute::gradientUnits, tag::value::userSpaceOnUse)
    { 
    m_gradientUnitsUserSpace = true; 
    }

void set(tag::attribute::gradientUnits, tag::value::objectBoundingBox)
    { 
    m_gradientUnitsUserSpace = false; 
    }
private:
    bool m_gradientUnitsUserSpace;
};

**<length>** or **<coordinate>** Is passed as single argument, whose type is set by Length Factory (by default double).

**<IRI>** or **<FuncIRI>** See IRI.

**<integer>** Single argument of int type is used.

**<number>** or **<opacity-value>** Is passed as single argument of number_type_ (by default double).

**<percentage>**

**<color>** Is passed as single argument, whose type is set by Color Factory (by default 8 bit per channel RGB packed in int).

**<color> **[<icccolor>]** If <icccolor> isn’t set, then it is passed as single argument, whose type is set by Color Factory. Otherwise, second argument is added, whose type is set by ICC Color Factory. Example:

```
struct Context
{ 
    void set(tag::attribute::flood_color, int rgb); 
    void set(tag::attribute::flood_color, int rgb, tag::skip_icc_color); 
    void set(tag::attribute::flood_color, tag::value::currentColor); 
    void set(tag::attribute::flood_color, tag::value::inherit); 
};
```

**<angle>** Is passed as single argument, whose type and value are set by Angle Factory (by default double value in degrees).

**<number-optional-number>** Is passed as one or two arguments of number_type_ type (by default double).

**<list-of-numbers>, <list-of-lengths> or <list-of-points>** Is passed as single argument of unspecified type, which is model of Boost Single Pass Range.

**range items have type:**

- number_type_ (by default double) in case of <list-of-numbers>;
- that is set by Length Factory in case of <list-of-lengths>;
- std::pair<number_type, number_type> (by default std::pair<double, double>) in case of <list-of-points>.

Example:
struct Context
{
    template<class Range>
    void set(tag::attribute::kernelMatrix, Range const & range)
    {
        for(typename boost::range_iterator<Range>::type it = boost::begin(range), end,
            it != end; ++it)
            std::cout << *it;
    }
};

Note: If template function can’t be used (e.g. it is virtual function), then Boost any_range can be used as range type instead:

typedef boost::any_range<double, boost::single_pass_traversal_tag, double const &,
std::ptrdiff_t> Range;

<shape> Is passed as 5 arguments - first is tag::value::rect, others are of number_type_ type (by default double): (tag::value::rect(), top, right, bottom, left).

viewBox attribute Is passed as 4 arguments of number_type_ type (by default double): (x, y, width, height).

bbox attribute Is passed as 4 arguments of number_type_ type (by default double): (lo_x, lo_y, hi_x, hi_y).

preserveAspectRatio attribute

Depending on value is passed as:

- (bool defer, tag::value::none)
- (bool defer, AlignT align, MeetOrSliceT meetOrSlice)

Type AlignT is one of tag::value::xMinYMin, tag::value::xMidYMin, tag::value::xMaxYMin, tag::value::xMinYMid, tag::value::xMidYMid, tag::value::xMaxYMid, tag::value::xMinYMax, tag::value::xMidYMax, tag::value::xMaxYMax. Type MeetOrSliceT is tag::value::meet or tag::value::slice.

text-decoration property none and inherit values are passed as Literal Values. Other values are passed as 8 arguments, 4 of which is of type bool, each of them preceded with tag, describing argument meaning. Boolean parameters takes true value if corresponding text decoration is set in property:

struct Context
{
    void set(tag::attribute::text_decoration, tag::value::none);
    void set(tag::attribute::text_decoration, tag::value::inherit);
    void set(tag::attribute::text_decoration, tag::value::underline, bool underline, tag::value::overline, bool overline, tag::value::line_through, bool line_through, tag::value::blink, bool blink);
};

enable-background property accumulate, new and inherit values are passed as Literal Values. Values as new <x> <y> <width> <height> are passed as 5 arguments, first of them is tag, other have type number_type_ (by
default: double): (tag::value::new_(), x, y, width, height).

<paint> Possible combinations of argument types:

- (tag::value::inherit)
- (tag::value::none)
- (tag::value::currentColor)
- (<color>)
- (<color>, <icccolor>)
- (<iri>, tag::value::none)
- (<iri>, tag::value::currentColor)
- (<iri>, <color>)
- (<iri>, <color>, <icccolor>)

Which types corresponds to <color> and <icccolor> is described above.

If IRI Policy policy::iri::distinguish_local is used, then number of methods with <iri> is doubled:

- (tag::value::inherit)
- (tag::value::none)
- (tag::value::currentColor)
- (<color>)
- (<color>, <icccolor>)
- (<iri>, tag::value::none)
- (tag::iri_fragment, <iri_fragment>, tag::value::none)
- (<iri>, tag::value::currentColor)
- (tag::iri_fragment, <iri_fragment>, tag::value::currentColor)
- (<iri>, <color>)
- (tag::iri_fragment, <iri_fragment>, <color>)
- (<iri>, <color>, <icccolor>)
- (tag::iri_fragment, <iri_fragment>, <color>, <icccolor>)

Example:

```cpp
typedef boost::variant<tag::value::none, tag::value::currentColor, int/* rgba */> SolidPaint;

struct IRIPaint
{
  IRIPaint(
    std::string const & fragment,
    boost::optional<SolidPaint> const & fallback = boost::optional<SolidPaint>(()));
};

typedef boost::variant<SolidPaint, IRIPaint> Paint;
```

(continues on next page)
template<class AttributeTag>
class PaintContext
{
public:
void set(AttributeTag, tag::value::none)
{
  m_paint = tag::value::none();
}

void set(AttributeTag, tag::value::currentColor)
{
  m_paint = tag::value::currentColor();
}

void set(AttributeTag, int color, tag::skip_icc_color = tag::skip_icc_color())
{
  m_paint = color;
}

template<class IRI>
void set(AttributeTag tag, IRI const & iri)
{
  throw std::runtime_error("Non-local references aren't supported");
}

template<class IRI>
void set(AttributeTag tag, tag::iri_fragment, IRI const & fragment)
{
  m_paint = IRIPaint(std::string(boost::begin(fragment), boost::end(fragment)));
}

template<class IRI>
void set(AttributeTag tag, IRI const & tag::value::none val)
{
  // Ignore non-local IRI, fallback to second option
  set(tag, val);
}

template<class IRI>
void set(AttributeTag tag, tag::iri_fragment, IRI const & fragment, _
  _tag::value::none val)
{
  m_paint = IRIPaint(std::string(boost::begin(fragment), boost::end(fragment)),
    boost::optional<SolidPaint>(val));
}

template<class IRI>
void set(AttributeTag tag, IRI const & tag::value::currentColor val)
{
  // Ignore non-local IRI, fallback to second option
  set(tag, val);
}

template<class IRI>
void set(AttributeTag tag, tag::iri_fragment, IRI const & fragment, _
  _tag::value::currentColor val)
{
m_paint = IRIPaint(std::string(boost::begin(fragment), boost::end(fragment)), boost::optional<SolidPaint>(val));
}

} // class IRI

void set(AttributeTag tag, IRI const & iri, agg::rgba8 val, tag::skip_icc_color = tag::skip_icc_color())
{
    // Ignore non-local IRI, fallback to second option
    set(tag, val);
}

} // class IRI

} // namespace

private:
    Paint m_paint;
};

1.4.3 Length

Length Factory defines what type corresponds to SVG types <length> and <coordinate> and how instances of that types are created from text values that include units.

Length Policy specifies how Length Factory for the context is accessed. It permits configuration of Length Factory in runtime (e.g. when viewport size or font size is changed).

Length Factory Concept

struct length_factory
{
    typedef /* ... */ length_type;
    typedef /* ... */ number_type;

    length_type create_length(number_type number, tag::length_units::em) const;
    length_type create_length(number_type number, tag::length_units::ex) const;
    length_type create_length(number_type number, tag::length_units::px) const;
    length_type create_length(number_type number, tag::length_units::in) const;
    length_type create_length(number_type number, tag::length_units::cm) const;
    length_type create_length(number_type number, tag::length_units::mm) const;
    length_type create_length(number_type number, tag::length_units::pt) const;
    length_type create_length(number_type number, tag::length_units::pc) const;
    length_type create_length(number_type number, tag::length_units::none) const;

    length_type create_length(number_type number, tag::length_dimension::width) const;
    length_type create_length(number_type number, tag::length_units::percent, tag::length_dimension::width) const;
    length_type create_length(number_type number, tag::length_units::percent, tag::length_dimension::height) const;

    (continues on next page)
create_length method receives number and length tag and returns corresponding length value of the type length_type.

Lengths that are set with percent units may be treated differently, depending on whether value corresponds to width or height. That is why create_length receives the third parameter - one of the three tag types tag::length_dimension::width, tag::length_dimension::height or tag::length_dimension::not_width_nor_height.

Depending on the implementation of Length Factory, length may be of integer type or something more complex like object containing value and units used. SVG++ provides factory::length::unitless, that implements configurable Length Factory returning numeric values.

**Unitless Length Factory**

Unitless Length Factory factory::length::unitless - is a model of Length Factory, provided by SVG++ library. Name “unitless” means that lengths created by factory are numbers without information about units. Unitless Length Factory uses units information to apply the corresponding coefficient to the value.

```cpp
template<
    class LengthType = double,
    class NumberType = double,
    class ReferenceAbsoluteUnits = tag::length_units::mm
>
class unitless
{
public:
    /* Skipped Length Factory methods */

template<class AbsoluteUnits>
    void set_absolute_units_coefficient(NumberType coeff, AbsoluteUnits);

template<class AbsoluteUnits>
    NumberType get_absolute_units_coefficient(AbsoluteUnits) const;

    void set_user_units_coefficient(NumberType coeff);
    NumberType get_user_units_coefficient() const;

    void set_viewport_size(LengthType width, LengthType height);

    void set_em_coefficient(NumberType coeff);
    void set_ex_coefficient(NumberType coeff);

template<class UnitsTag>
    void set_em_coefficient(NumberType coeff, UnitsTag unitsTag);

template<class UnitsTag>
    void set_ex_coefficient(NumberType coeff, UnitsTag unitsTag);
};
```

**set_absolute_units_coefficient** Sets coefficient that will be used for lengths in absolute units (in, cm, mm, pt or pc). The method should be called for any one of absolute units, coefficients for others will be calculated automatically. Example:

```cpp
length_type create_length(number_type number, tag::length_units::percent,
    tag::length_dimension::not_width_nor_height) const;
```
// Let our length value be a pixel. Set 'in' coefficient to 90 (90 Dots per inch)
factory.set_absolute_units_coefficient(90, svgpp::tag::length_units::in());
// Coefficient for 'pc' (pica = 1/6 inch) will be 90/6 = 15
assert(factory.get_absolute_units_coefficient(svgpp::tag::length_units::pc()) == 15);

**set_user_units_coefficient** Sets coefficient that will be used for lengths in user units (that are specified without units or in `px`). By default coefficient = 1.

**set_viewport_size** Sets width and height of the viewport to be used for percentage values that are defined to be relative to the size of viewport.

**set_em_coefficient** and **set_ex_coefficient** Sets coefficients for font-related `em` and `ex` units.

### Length Policy Concept

**Length Policy** defines which **Length Factory** will be used for the context:

```cpp
struct length_policy
{
    typedef /* ... */ length_factory_type;
    static length_factory_type & length_factory(context_type & context);
};
```

Default **Length Policy** returns constant reference to static instance of `factory::length::default_factory`. 

**Named class template parameter** for **Length Policy** is `length_policy`.

To configure **Length Factory** named template parameter `length_policy` must be passed to `document_traversal`. For example, using library provided `policy::length::forward_to_method`:

```cpp
typedef factory::length::unitless<> LengthFactory;

class Context
{
public:
    LengthFactory const & length_factory() { return m_LengthFactory; }

private:
    LengthFactory m_LengthFactory;
};

document_traversal<
    length_policy<policy::length::forward_to_method<Context, LengthFactory const> >,
    /* ... */
    >::/* ... */
```

### 1.4.4 Transform

Transformation parsing is controlled by **Transform Policy** and **Transform Events Policy**. **Transform Policy** configures the adapter that simplifies application handling of coordinate system transformations (e.g. adapter may substitute simple transformation steps translate/scale/rotate/skew with corresponding transformation matrices, more convenient in some cases). **Transform Events Policy** defines how parsed data is passed to the user code.
Transform Events Policy Concept

Transform Events Policy depends on compile-time settings in Transform Policy. If policy::transform::raw Transform Policy is used that preserves input data at much as possible then Transform Events Policy becomes:

```cpp
struct transform_events_policy
{
    typedef /*...*/ context_type;

    static void transform_matrix(context_type & context, const boost::array<number_type, 6> & matrix);
    static void transform_translate(context_type & context, number_type tx, number_type ty);
    static void transform_translate(context_type & context, number_type tx);
    static void transform_scale(context_type & context, number_type sx, number_type sy);
    static void transform_scale(context_type & context, number_type scale);
    static void transform_rotate(context_type & context, number_type angle);
    static void transform_rotate(context_type & context, number_type angle, number_type cx, number_type cy);
    static void transform_skew_x(context_type & context, number_type angle);
    static void transform_skew_y(context_type & context, number_type angle);
};
```

Depending on Transform Policy, some of the methods aren’t called by SVG++ and therefore shouldn’t be implemented.

Note: Transformation matrix is passed as array of size 6 \([a \ b \ c \ d \ e \ f]\), corresponding to this matrix:

```
\[
\begin{bmatrix}
  a & c & e \\
  b & d & f \\
  0 & 0 & 1
\end{bmatrix}
```

Named class template parameter for Transform Events Policy is transform_events_policy.

Default Transform Events Policy (policy::transform_events::forward_to_method) forwards calls to its static methods to methods of context object:

```cpp
struct forward_to_method
{
    typedef Context context_type;

    template<
        class Number
    >
    static void transform_matrix(context_type & context, const boost::array<Number, 6> & matrix)
    {
        context.transform_matrix(matrix);
    }

    /*...*/
};
```

Example of handling transforms with default settings (src/samples/sample_transform01.cpp):
```cpp
#include <svgpp/svgpp.hpp>
#include <algorithm>
#include <iterator>

using namespace svgpp;

struct Context {
    void transform_matrix(const boost::array<double, 6> & matrix) {
        std::copy(matrix.begin(), matrix.end(),
                   std::ostream_iterator<double>(std::cout, " "));
        std::cout << "\n";
    }
};

int main() {
    Context context;
    value_parser<tag::type::transform_list>::parse(tag::attribute::transform(), context,
        std::string("translate(-10,-20) scale(2) rotate(45) translate(5,10)"),
        tag::source::attribute());
    return 0;
}
```

In this example sequential transforms are joined in user code (src/samples/sample_transform02.cpp):

```cpp
#include <svgpp/svgpp.hpp>
#include <boost/version.hpp>
#if BOOST_VERSION >= 106400
#include <boost/serialization/array_wrapper.hpp>
#endif
#include <boost/math/constants/constants.hpp>
#include <boost/numeric/ublas/assignment.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>

using namespace svgpp;
namespace ublas = boost::numeric::ublas;

typedef ublas::matrix<double> matrix_t;

struct TransformEventsPolicy {
    typedef matrix_t context_type;

    static void transform_matrix(matrix_t & transform, const boost::array<double, 6> & matrix) {
        matrix_t m(3, 3);
        m <<=
            matrix[0], matrix[2], matrix[4],
            matrix[1], matrix[3], matrix[5],
            0, 0, 1;
        transform = ublas::prod(transform, m);
    }
};
```

(continues on next page)
static void transform_translate(matrix_t & transform, double tx, double ty)
{
    matrix_t m = ublas::identity_matrix<double>(3, 3);
    m(0, 2) = tx; m(1, 2) = ty;
    transform = ublas::prod(transform, m);
}
static void transform_scale(matrix_t & transform, double sx, double sy)
{
    matrix_t m = ublas::identity_matrix<double>(3, 3);
    m(0, 0) = sx; m(1, 1) = sy;
    transform = ublas::prod(transform, m);
}
static void transform_rotate(matrix_t & transform, double angle)
{
    angle *= boost::math::constants::degree<double>();
    matrix_t m(3, 3);
    m <<=
        std::cos(angle), -std::sin(angle), 0,
        std::sin(angle), std::cos(angle), 0,
        0, 0, 1;
    transform = ublas::prod(transform, m);
}
static void transform_skew_x(matrix_t & transform, double angle)
{
    angle *= boost::math::constants::degree<double>();
    matrix_t m = ublas::identity_matrix<double>(3, 3);
    m(0, 1) = std::tan(angle);
    transform = ublas::prod(transform, m);
}
static void transform_skew_y(matrix_t & transform, double angle)
{
    angle *= boost::math::constants::degree<double>();
    matrix_t m = ublas::identity_matrix<double>(3, 3);
    m(1, 0) = std::tan(angle);
    transform = ublas::prod(transform, m);
}
};

int main()
{
    matrix_t transform(ublas::identity_matrix<double>(3, 3));
    value_parser<
        tag::type::transform_list,
        transform_policy<policy::transform::minimal>,
        transform_events_policy<TransformEventsPolicy>
    >::parse(tag::attribute::transform(), transform,
        std::string("translate(-10,-20) scale(2) rotate(45) translate(5,10)"),
        tag::source::attribute());
    std::cout << transform << "\n";
    return 0;
}
Transform Policy Concept

```cpp
struct transform_policy_concept
{
    static const bool join_transforms = /* true or false */;
    static const bool no_rotate_about_point = /* true or false */;
    static const bool no_shorthands = /* true or false */;
    static const bool only_matrix_transform = /* true or false */;
};
```

`Transform Policy` is a class with `bool` static member constants. If they all are `false` (as in `policy::transform::raw`), then adapter isn’t used and the parser passes parsed values as is. Setting some members to `true` programmer may simplify the application:

- `join_transforms = true` All transformations in SVG attribute are joined in single transformation matrix. Values of other `Transform Policy` members are ignored. `Transform Events Policy` in this case contains only one method:

  ```cpp
  struct transform_events_policy
  {
      typedef /*...*/ context_type;
      static void transform_matrix(context_type & context, const boost::array<
  ```

  - `no_rotate_about_point = true` `rotate(<rotate-angle> <cx> <cy>)` substituted with `translate(<cx>, <cy>) rotate(<rotate-angle>) translate(-<cx>, -<cy>)`. `transform_rotate` method of `Transform Events Policy` with parameters `cx` and `cy` is not used.

  - `no_shorthands = true` Calls to `transform_translate` and `transform_scale` with one number substituted with corresponding calls with two numbers.

  - `only_matrix_transform = true` Each transformation step is substituted with corresponding call to `transform_matrix`. Therefore only this method is used in `Transform Events Policy`.

File `svgpp/policy/transform.hpp` contains some predefined `Transform Policies`. `policy::transform::matrix`, used by default, sets `join_transforms = true`.

Named class template parameter for `Transform Policy` is `transform_policy`.

1.4.5 Path

Path parsing is controlled by `Path Policy` and `Path Events Policy`. `Path Policy` configures the adapter that simplifies path handling for application (e.g. adapter may substitute relative coordinates with absolute ones relieving programmer from this duty). `Path Events Policy` defines how parsed path data is passed to the user code.

Path Events Policy Concept

```cpp
struct path_events_concept
{
    typedef /*...*/ context_type;

    static void path_move_to(context_type & context, number_type x, number_type y, ...
  ```

(continues on next page)
static void path_line_to(context_type & context, number_type x, number_type y,
   →AbsoluteOrRelative absoluteOrRelative);
static void path_line_to_ortho(context_type & context, number_type coord, bool,
→horizontal, AbsoluteOrRelative absoluteOrRelative);
static void path_cubic_bezier_to(context_type & context, number_type x1, number_-
→type y1,
   number_type x2, number_type y2,
   number_type x, number_type y,
   AbsoluteOrRelative absoluteOrRelative);
static void path_cubic_bezier_to(context_type & context,
   number_type x2, number_type y2,
   number_type x, number_type y,
   AbsoluteOrRelative absoluteOrRelative);
static void path_quadratic_bezier_to(context_type & context,
   number_type x1, number_type y1,
   number_type x, number_type y,
   AbsoluteOrRelative absoluteOrRelative);
static void path_quadratic_bezier_to(context_type & context,
   number_type x, number_type y,
   AbsoluteOrRelative absoluteOrRelative);
static void path_elliptical_arc_to(context_type & context,
   number_type rx, number_type ry, number_type x_,
→axis_rotation,
   bool large_arc_flag, bool sweep_flag,
   number_type x, number_type y,
   AbsoluteOrRelative absoluteOrRelative);
static void path_close_subpath(context_type & context);
static void path_exit(context_type & context);
}

absoluteOrRelative parameter may have type tag::coordinate::absolute or
tag::coordinate::relative depending on whether absolute or relative coordinates are passed in other
parameters.

Depending on Path Policy, some of the methods aren’t called by SVG++ and therefore shouldn’t be implemented.

Default Path Events Policy (policy::path_events::forward_to_method) forwards calls to its static meth-
ods to context object methods:

struct forward_to_method
{
  typedef Context context_type;

  template<class Coordinate, class AbsoluteOrRelative>
  static void path_move_to(Context & context, Coordinate x, Coordinate y,
   →AbsoluteOrRelative absoluteOrRelative)
  {
    context.path_move_to(x, y, absoluteOrRelative);
  }

  /*...*/
};

Named class template parameter for Path Events Policy is path_events_policy.
Path Policy Concept

```cpp
struct path_policy_concept {
    static const bool absolute_coordinates_only = /* true or false */;
    static const bool no_ortho_line_to = /* true or false */;
    static const bool no_quadratic_bezier_shorthand = /* true or false */;
    static const bool no_cubic_bezier_shorthand = /* true or false */;
    static const bool quadratic_bezier_as_cubic = /* true or false */;
    static const bool arc_as_cubic_bezier = /* true or false */;
};
```

Path Policy is a class with bool static member constants. If they all are false (as in `policy::path::raw`), then adapter isn’t used and parser passed parsed values to the user code as is. By setting some members to true programmer may simplify his work:

- **absolute_coordinates_only = true** Relative coordinates are replaced with corresponding absolute. Therefore Path Events Policy methods with `tag::coordinate::relative` parameter aren’t used.

- **no_ortho_line_to = true** Instead of call to `path_line_to_ortho` with one coordinate call is made to `path_line_to` with two coordinates.

- **no_quadratic_bezier_shorthand = true** Instead of `path_quadratic_bezier_to` with two coordinates (shorthand/smooth curve) overload of same method with four coordinates is used.

- **no_cubic_bezier_shorthand = true** Instead of `path_cubic_bezier_to` with four coordinates (shorthand/smooth curve) overload of same method with six coordinates is used.

- **quadratic_bezier_as_cubic = true** Call to `path_quadratic_bezier_to` is replaced with call to `path_cubic_bezier_to`.

- **arc_as_cubic_bezier = true** Elliptical arc is approximated with cubic Bézier curve. Call to `path_elliptical_arc_to` substituted with series of calls to `path_cubic_bezier_to`.

Named class template parameter for Path Policy is `path_policy`.

File `svgpp/policy/path.hpp` contains some predefined Path Policies. `policy::path::no_shorthands` used by default limits Path Events Policy interface as much as possible not using approximation. In this case Path Events Policy becomes:

```cpp
struct path_events_no_shorthands_concept {
    typedef /*...*/ context_type;

    static void path_move_to(context_type & context, number_type x, number_type y, tag::coordinate::absolute);
    static void path_line_to(context_type & context, number_type x, number_type y, tag::coordinate::absolute);
    static void path_cubic_bezier_to(context_type & context, number_type x1, number_type y1, number_type x2, number_type y2, number_type x, number_type y, tag::coordinate::absolute);
    static void path_quadratic_bezier_to(context_type & context, number_type x1, number_type y1, number_type x, number_type y, tag::coordinate::absolute);
}(continues on next page)


It is better to inherit own *Path Policy* from some provided by SVG++ to easy upgrade to future versions of SVG++.

### 1.4.6 Basic Shapes

Basic shapes (*rect*, *circle*, *ellipse*, *line*, *polyline* and *polygon*) can be automatically converted to *path* commands.

#### Basic Shapes Policy Concept

```cpp
struct basic_shapes_policy
{
    typedef /* Associative Sequence */ convert_to_path;
    typedef /* Associative Sequence */ collect_attributes;

    static const bool convert_only_rounded_rect_to_path /* true or false */;
};
```

*convert_to_path* Associative Sequence (e.g. boost::mpl::set), containing SVG *basic shapes* elements tags, which will be converted to *path*. Generated *path* uses *Path Policy* and *Path Events Policy* settings.

*collect_attributes* Associative Sequence, containing any of *rect*, *circle*, *ellipse* or *line* element tags. Geometry of elements included in the sequence will be passed with single call instead of separate attribute handling (see *Basic Shapes Events Policy*).

*convert_only_rounded_rect_to_path* If static member constant *convert_only_rounded_rect_to_path* equals to *true* and *tag::element::rect* is included in *convert_to_path* sequence, then only rounded rectangles will be converted to *path*, and regular rectangles will be handle like if *tag::element::rect* is included in *collect_attributes*.

In document traversal processing of attributes describing *basic shapes* geometry (*x*, *y*, *r* etc) must be enabled, i.e. they must be included in *processed_attributes* or excluded from *ignored_attributes*. Associative Sequence traits::shapes_attributes_by_element contains tags of all such attributes for *basic shapes*.

#### Basic Shapes Events Policy Concept

```cpp
struct basic_shapes_events_policy
{
    static void set_rect(Context & context, Coordinate x, Coordinate y, Coordinate width, Coordinate height);
    static void set_rect(Context & context, Coordinate x, Coordinate y, Coordinate width, Coordinate height, Coordinate rx, Coordinate ry);
};
```

(continues on next page)
static void set_line(Context & context, Coordinate x1, Coordinate y1, Coordinate x2, Coordinate y2);
static void set_circle(Context & context, Coordinate cx, Coordinate cy, Coordinate r);
static void set_ellipse(Context & context, Coordinate cx, Coordinate cy, Coordinate rx, Coordinate ry);
};

Basic Shapes Events Policy is used for basic shapes (except polyline and polygon) elements, that are listed in collect_attributes field of Basic Shapes Policy.

Adapters that implement these conversions, use length_to_user_coordinate method of Length Factory to get user coordinates value by length. These adapters passes default values if attributes are missing and check correctness of attributes. If value disables rendering of the element according to SVG specification, then Basic Shapes Events Policy methods aren’t called, and if an attribute has negative value that is not permitted by specification, then negative_value function of Error Policy is called.

Default Basic Shapes Events Policy (policy::basic_shapes_events::forward_to_method) forwards calls to its static methods to context object methods:

struct forward_to_method
{
    template<class Coordinate>
    static void set_rect(Context & context, Coordinate x, Coordinate y, Coordinate width, Coordinate height, Coordinate rx, Coordinate ry)
    {
        context.set_rect(x, y, width, height, rx, ry);
    }
/*...*/
};

Named class template parameter for Basic Shapes Events Policy is basic_shapes_events_policy.

1.4.7 Color

Color Factory defines what type corresponds to <color> SVG type and how values of this type are created from SVG color description.

ICC Color Factory defines what type corresponds to <icccolor> SVG type and how values of this type are created from SVG ICC color description.

Color Factory Concept

struct color_factory
{
    typedef /* ... */ color_type;
    typedef /* floating point number type */ percentage_type;

    static color_type create(unsigned char r, unsigned char g, unsigned char b);
    static color_type create_from_percent(percentage_type r, percentage_type g, percentage_type b);
};
create_from_percent is called when color components are set as percents (e.g. \texttt{rgb(100\%,100\%,100\%)})
Percent values are passed as is, i.e. 100\% is passed as 100.

In other cases 
\texttt{create} 
function is called with integer component values in range 0 to 255. Recognized color keyword
names are converted to corresponding component values by SVG++ library.

System colors aren’t handled yet.

\textit{Named class template parameter} for \texttt{Color Factory} is \texttt{color\_factory}.

\textbf{Integer Color Factory}

SVG++ by default uses \texttt{factory::color::integer<>} as \texttt{Color Factory}. This factory returns color packed in
\texttt{int} value: Red component in 3rd byte, Green in 2nd, Blue in 1st (least significant). Component offsets and number
type can be configured.

\textbf{ICC Color Factory Concept}

\begin{verbatim}
struct icc_color_factory
{
    typedef /* floating point number type */ component_type;
    typedef /* ... */ icc_color_type;
    typedef /* ... */ builder_type;

    template<class Iterator>
    void set_profile_name(builder_type &),
    typename boost::iterator_range<Iterator> const &)
    const;
    void append_component_value(builder_type &,
    component_type) const;

    icc_color_type create_icc_color(builder_type const &) const;
};
\end{verbatim}

\texttt{icc\_color\_type} is a type that will be passed to the user code.

\texttt{builder\_type} is used as a temporary object during building \texttt{icc\_color\_type} from color profile name and
some components values.

Pseudo-code that illustrates usage of \textit{ICC Color Factory} from inside SVG++ to parse \texttt{icc\_color(profile1 0.75, 0.15, 0.25)} value:

\begin{verbatim}
void parse_icc(icc_color_factory const & factory)
{
    icc_color_factory::builder_type builder;
    factory.set_profile_name(builder, boost::as_literal("profile1"));
    factory.append_component_value(builder, 0.75);
    factory.append_component_value(builder, 0.15);
    factory.append_component_value(builder, 0.25);
    value_events_policy::set(context, attribute_tag, factory.create_icc_color(builder));
}
\end{verbatim}

\textbf{ICC Color Policy Concept}

\textit{ICC Color Policy} defines which \textit{ICC Color Factory} instance is used for the context. It can be used to configure \textit{ICC Color Factory} in runtime, e.g. taking in account referenced color profiles in SVG document.


```cpp
struct icc_color_policy
{
  typedef /* ... */ context_type;
  typedef /* ... */ icc_color_factory_type;

  static icc_color_factory_type & icc_color_factory(Context const &);
};
```

Default **ICC Color Policy** returns static instance of `factory::icc_color::stub`. `factory::icc_color::stub` is a model of **ICC Color Factory** that skips passed values and returns instance of empty `tag::skip_icc_color` class as ICC color.

**Named class template parameter** for **ICC Color Policy** is `icc_color_policy`.

### 1.4.8 Error handling

**Returned values**

In most cases when SVG++ methods returns `bool` values, they can be used as an alternative to exceptions - if method returns `false`, then calling method immediately returns with `false` result and so on.

Error reporting is controlled by **Error Policy**. `policy::error::raise_exception` used by default throws exception objects, derived from `std::exception` and `boost::exception`. In this case only `true` may be returned as a result code.

**Default Error Handling**

Default `policy::error::raise_exception` uses Boost.Exception for transporting arbitrary data to the catch site.

`boost::error_info` uses tags `tag::error_info::xml_element` and `tag::error_info::xml_attribute` to pass information about place in SVG document where error occurred alongside with the exception object. Value type depends on XML parser and `XML Policy` used.

Example of SVG++ exception handling when RapidXML NS parser is used:

```cpp
typedef rapidxml_ns::xml_node<> const * XMLElement;

typedef boost::error_info<svgpp::tag::error_info::xml_element, XMLElement> element_error_info;

try
{
  /* ... */
  document_traversal</* ... */>::load_document</* ... */>();
}
catch(svgpp::exception_base const & e)
{
  typedef boost::error_info<svgpp::tag::error_info::xml_element, XMLElement> element_error_info;
  std::cerr << "Error reading SVG";
  if((XMLElement const * element = boost::get_error_info<element_error_info>(e)))
  {
    std::cerr << " in element '" << std::string((*element)->name(), (*element)->name() + (*element)->name_size()) << ": " << e.what() << "\n";
  }
}
### Error Policy Concept

```cpp
struct error_policy
{
    typedef /* ... */ context_type;

template<class XMLElement, class ElementName>
static bool unknown_element(
    context_type const &,
    XMLElement const & element,
    ElementName const & name);

template<class XMLAttributesIterator, class AttributeName>
static bool unknown_attribute(context_type &,
    XMLAttributesIterator const & attribute,
    AttributeName const & name,
    BOOST_SCOPED_ENUM(detail::namespace_id) namespace_id,
    tag::source::attribute);

template<class XMLAttributesIterator, class AttributeName>
static bool unknown_attribute(context_type &,
    XMLAttributesIterator const & attribute,
    AttributeName const & name,
    tag::source::css);
static bool unexpected_attribute(context_type &,
    detail::attribute_id id, tag::source::attribute);

template<class AttributeTag>
static bool required_attribute_not_found(context_type &, AttributeTag);

template<class AttributeTag, class AttributeValue>
static bool parse_failed(context_type &, AttributeTag,
    AttributeValue const & value);

template<class XMLElement>
static bool unexpected_element(context_type &, XMLElement const & element);

template<class AttributeTag>
static bool negative_value(context_type &, AttributeTag);

typedef /* ... */ intercepted_exception_type;

template<class XMLElement>
static bool add_element_info(intercepted_exception_type & e,
    XMLElement const & element);
};
```

If *Error Policy* method returns `true`, then SVG++ continues SVG processing skipping the part with the error. In some cases it may lead to problems in further processing.

If method returns `false`, then processing immediately stops.

### 1.4.9 IRI

IRIs, including ones that are set with FuncIRI syntax, are passed to the user code the same way as other *strings* in SVG++. 

1.4. Library Reference
The only setting in *IRI Policy* configures whether to distinguish local references to document fragment (IRIs prefixed with “#”) from non-local IRI references. If `policy::iri::distinguish_local` used (as by default), then local reference to document fragment is passed as pair of parameters: `{tag::iriFragment(), <fragment string>}`. If `policy::iri::raw` is set, then any IRI is passed as single `string`.

**Named class template parameter** for *IRI Policy* is `iri_policy`.

Example of using default *IRI Policy* (`src/samples/sample_iri.cpp`):

```cpp
#include <svgpp/svgpp.hpp>
using namespace svgpp;

struct Context
{
    template<class String>
    void set(tag::attribute::xlink::href, String const & iri)
    {
        std::cout << "IRI: " << std::string(boost::begin(iri), boost::end(iri)) << "\n";
    }

    template<class String>
    void set(tag::attribute::xlink::href, tag::iri_fragment, String const & iri)
    {
        std::cout << "Fragment: " << std::string(boost::begin(iri), boost::end(iri)) << "\n";
    }
};

int main()
{
    Context context;
    value_parser<tag::type::iri>::parse(tag::attribute::xlink::href(), context,
        std::string("http://foo.com/bar#123"), tag::source::attribute());
    value_parser<tag::type::iri>::parse(tag::attribute::xlink::href(), context,
        std::string("#rect1"), tag::source::attribute());
    // Output:
    // "IRI: http://foo.com/bar#123"
    // "Fragment: rect1"
    return 0;
}
```

### 1.4.10 Markers

SVG++ provides option to automatically calculate marker symbols orientation on lines (`path`, `line`, `polyline` or `polygon` element).

*Markers Policy* turns on/off and configures marker position calculations. *Marker Events Policy* defines how marker positions are passed to the user code.

**Markers Policy Concept**

```cpp
struct makers_policy_concept
{
    typedef /* ... */ directionality_policy;
}
```

(continues on next page)
calculate_markers = true Enables marker position calculations. If calculate_markers = false, then other class members aren’t used.

Marker properties define which vertices contain markers. Depending on the value of orient attribute of marker element, marker orientation may be fixed or calculated by line geometry (auto value).

always_calculate_auto_orient = true In this case marker orientation is calculated for each vertex (orientation required in orient=”auto” case).

always_calculate_auto_orient = false In this case, before processing each line element, marker_get_config method of Marker Events Policy is called, to request from user code which vertices markers are required (user code should know this from marker properties). Marker positions are returned by Marker Events Policy only for vertices, for which user code requested marker calculations.

directionality_policy Class that defines how marker orientation is calculated and passed. By default marker orientation is double value, containing angle in radians.

File svgpp/policy/markers.hpp contains some predefined Markers Policies: policy::markers::calculate_always, policy::markers::calculate and policy::markers::raw. policy::markers::raw used by default disables automatic marker calculation.

Named class template parameter for Markers Policy is markers_policy.

Marker Events Policy Concept

```cpp
namespace svgpp {

enum marker_vertex { marker_start, marker_mid, marker_end };
enum marker_config { marker_none, marker_orient_fixed, marker_orient_auto };
}
```

```cpp
struct marker_events_policy_concept {

typedef /* ... */ context_type;

static void marker_get_config(context_type & context, 
marker_config & start, marker_config & mid, marker_config & end);

static void marker(context_type & context, marker_vertex v, 
number_type x, number_type y, directionality_type directionality, unsigned marker_˓→index);

static void marker(context_type & context, marker_vertex v, 
number_type x, number_type y, tag::orient_fixed directionality, unsigned marker_˓→index);
};
```

marker_get_config method is called if makers_policy::always_calculate_auto_orient = false. User code must set for each vertex type (start, mid and end), whether it wants to calculate orientation and position (marker_orient_auto), position only (marker_orient_fixed) or neither of them (marker_none).
marker method is called for each vertex calculated with this arguments:

- \( x \) and \( y \) - marker position
- \( v \) - vertex type (marker_start, marker_mid or marker_end)
- **directionality** - marker orientation (by default double value in radians). If marker_get_config returned marker_orient_fixed for this type of vertex, then tag tag::orient_fixed is passed instead of orientation value.

**marker_index** Marker orientations aren’t always calculated and passed in the order, because orientation of marker on first vertex of subpath may be calculated only after subpath is finished. marker_index is the 0-based index of the marker in drawing order.

**Named class template parameter** for Marker Events Policy is marker_events_policy.

Default **Marker Events Policy**(policy::marker_events::forward_to_method) forwards calls to its static methods to context object methods.

Example:

```cpp
class Context
{
  public:
    void marker(svgpp::marker_vertex v, double x, double y, double directionality, unsigned marker_index)
    {
      if (markers_.size() <= marker_index)
        markers_.resize(marker_index + 1);
      auto & m = markers_[marker_index];
      m.v = v;
      m.x = x;
      m.y = y;
      m.directionality = directionality;
    }

  private:
    struct MarkerPosition
    {
      svgpp::marker_vertex v;
      double x, y, directionality;
    };

    std::vector<MarkerPosition> markers_;
  };

  void load()
  {
    document_traversal<
      /* ... */
      svgpp::markers_policy<svgpp::policy::markers::calculate_always>
    >::load_document(xml_root_element, context);
  }
}

1.4.11 Viewport

When establishing new viewport by elements svg, symbol (instanted by use) or image several attributes must be processed (x, y, width, height, preserveAspectRatio, viewBox) to get the new user coordinate system, viewport and
clipping path. SVG++ may do it itself and present result in convenient way. This behavior is configured by Viewport Policy. Viewport Events Policy defines how viewport information is passed to the user code.

**marker** element establish new viewport similar way, when it is instances in path vertices, so **marker** elements may also be handled by Viewport Policy and Viewport Events Policy.

### Viewport Policy Concept

```cpp
struct viewport_policy_concept {
    static const bool calculate_viewport = /* true or false */;
    static const bool calculate_marker_viewport = /* true or false */;
    static const bool calculate_pattern_viewport = /* true or false */;
    static const bool viewport_as_transform = /* true or false */;
};
```

**calculate_viewport** = true Enables handling of viewport attributes for **svg** and **symbol** elements. Values will be passed by calls to **set_viewport** or **set_viewbox_transform** of Viewport Events Policy.

**calculate_marker_viewport** = true The same for **marker** elements.

**calculate_pattern_viewport** = true The same for **pattern** elements.

**viewport_as_transform** = true Is checked only if **calculate_viewport** = true. 
**set_viewbox_transform** method of Viewport Events Policy isn’t used, new coordinate system is set by Transform Events Policy instead.

Named class template parameter for Viewport Policy is viewport_policy.

File *svgpp/policy/viewport.hpp* contains some predefined Viewport Policies. 
**policy::viewport::raw**, used by default, turns off viewport processing in SVG++.
**policy::viewport::as_transform** sets all boolean members to true, thus enabling viewport attributes handling by SVG++ and passing coordinate system changes by Transform Events Policy.

### Viewport Events Policy Concept

```cpp
struct viewport_events_policy_concept {
    static void set_viewport(context_type & context, number_type viewport_x, number_type viewport_y,
                             number_type viewport_width, number_type viewport_height);

    static void set_viewbox_transform(context_type & context, number_type translate_x, number_type translate_y,
                                       number_type scale_x, number_type scale_y, bool defer);

    static void set_viewbox_size(context_type & context, number_type viewbox_width, number_type viewbox_height);

    static void get_reference_viewport_size(context_type & context, number_type & viewport_width, number_type & viewport_height);

    static void disable_rendering(context_type & context);
};
```

**set_viewport** Sets new viewport position.
set_viewbox_transform  Passes offset and scale that are set by combination of viewBox and preserveAspectRatio. Not used if viewport_as_transform = true.

set_viewbox_size  Passes size of viewBox, set by viewBox attribute. Called only if viewBox attribute is present. It can be used to set viewport size in Length Factory.

get_reference_viewport_size  See Referenced element.

disable_rendering  SVG Specification says that “[width or height attribute] value of zero disables rendering of this element”. disable_rendering will be called in such cases, so that application can behave in proper way. Document Traversal Control Policy may be used to skip processing of element content.

Named class template parameter for Viewport Events Policy is viewport_events_policy.

Referenced element

When processing svg element referenced by use or image, symbol referenced by use values of width and height of referencing element are required.

To pass to SVG++ information that element is referenced by use or image, tag of referenced element must be passed as referencing_element parameter of document_traversal class. In this case get_reference_viewport_size method of Viewport Events Policy will be called. Implementation of get_reference_viewport_size must set viewport_width to value of width attribute of referenced element, if this attribute is set. And set viewport_height value of height attribute of referenced element, if this attribute is set.

Processed attributes

If calculate_viewport = true or calculate_pattern_viewport = true in Viewport Policy, then SVG++ intercepts and processes attributes, listed in traits::viewport_attributes:

```cpp
namespace traits
{
    typedef boost::mpl::set6<
        tag::attribute::x,
        tag::attribute::y,
        tag::attribute::width,
        tag::attribute::height,
        tag::attribute::viewBox,
        tag::attribute::preserveAspectRatio
    > viewport_attributes;
}
```

If calculate_marker_viewport = true in Viewport Policy, then SVG++ intercepts and processes marker attributes listed in traits::marker_viewport_attributes:

```cpp
namespace traits
{
    typedef boost::mpl::set6<
        tag::attribute::refX,
        tag::attribute::refY,
        tag::attribute::markerWidth,
        tag::attribute::markerHeight,
        tag::attribute::viewBox,
        tag::attribute::preserveAspectRatio
    > marker_viewport_attributes;
}
```
Processing of this attributes must be enabled by the programmer.

**Order of viewport processing**

Viewport attributes will be processed and result will be passed by Viewport Events Policy after all SVG element attributes are processed or when notification with tag tag::event::after_viewport_attributes arrives.

**1.4.12 Character Data**

Character data content of elements, that supports it according to SVG Specification (text, desc, title, style etc), is passed to the user code by the Text Events Policy.

```cpp
struct text_events_policy_concept
{
    template<class Range>
    static void set_text(Context & context, Range const & text);
};
```

It is not guaranteed that sequential blocks of text will be joined together. set_text is called for each TEXT or CDATA node returned by XML parser.

**1.5 Advanced Topics**

**1.5.1 Adding new XML parser**

Implementing support for the new XML parser requires creating new specializations for svgpp::policy::xml::attribute_iterator and svgpp::policy::xml::element_iterator class templates using one of existing implementations in folder include/svgpp/policy/xml as a sample.

Understanding following types may cause some difficulties:

```cpp
template<>
struct attribute_iterator<CustomXMLAttribute>
{
    typedef /* ... */ string_type;
    typedef /* ... */ attribute_name_type;
    typedef /* ... */ attribute_value_type;
    typedef /* ... */ saved_value_type;

    /* ... */
};
```

**string_type** A model of Forward Range, with items of some character type (char, wchar_t, char16_t, char32_t). Easy solution is to use boost::iterator_range<CharT const * >.

**attribute_name_type** Copy constructible type, for which exists method string_type attribute_iterator::get_string_range(attribute_name_type const &). Value of attribute_name_type is used only before attribute_iterator gets incremented and may become invalid after that.

**attribute_value_type** Copy constructible type, for which exists method string_type attribute_iterator::get_string_range(attribute_value_type const &). Value of attribute_value_type must be accessible independently of attribute_iterator state changes.
**saved_value_type** Copy constructible type, for which exists method `attribute_value_type_attribute_iterator::get_value(saved_value_type const &)`. Object of this type must as efficiently as possible save attribute value that may be not requested. For example XML parser provides access to XML attribute like this:

```cpp
class XmlAttribute
{
public:
    std::string getValue() const;
};
```

In this case `saved_value_type` may be defined as `XmlAttribute const *`, instead of `std::string`, to avoid expenses of creating and coping string that may not be requested later.

### 1.5.2 Value Parsers

To choose proper `value_parser` for the attribute, `traits::attribute_type` metafunction should be used:

```cpp
template<class Element, class Attribute>
struct attribute_type
{
    typedef /*unspecified*/ type;
};
```

The returned type can be:

- One of type tags. E.g. `width`, as many others attributes has `<length>` type (corresponds to `tag::type::length`):

```cpp
BOOST_ASSERT(( boost::is_same<
    traits::attribute_type<tag::element::rect, tag::attribute::width>::type,
    tag::type::length> ));
```

- Attribute tag. E.g. `viewBox` has its own syntax:

```cpp
BOOST_ASSERT(( boost::is_same<
    traits::attribute_type<tag::element::svg, tag::attribute::viewBox>::type,
    tag::attribute::viewBox> ));
```

- Pair `<element tag, attribute tag>`. E.g. `type` attribute may get different values in elements `animateTransform`, `feColorMatrix`, `feTurbulence` etc.:

```cpp
BOOST_ASSERT(( boost::is_same<
    traits::attribute_type<tag::element::animateTransform, tag::attribute::type> ->::type,
    boost::mpl::pair<tag::element::animateTransform, tag::attribute::type> > ));
```

Attribute value parsers interface:

```cpp
template<class ValueType, class Args..>
struct value_parser
{
    template<class AttributeTag, class Context, class AttributeValue, class
    PropertySource>
    static bool parse(AttributeTag tag, Context & context, AttributeValue const &
    attribute_value,
    (continues on next page)
AttributeTag \texttt{tag} is passed to context, isn’t used by \texttt{value\_parser} itself

\texttt{Context \& context} Reference to the context that will \textit{receive} parsed value.

\texttt{AttributeValue const \& attribute\_value} Attribute \textit{string} value

\texttt{PropertySource source} One of two possible types: \texttt{tag::source::attribute} or \texttt{tag::source::css}, depending on whether value if from SVG attribute or CSS property.

## 1.6 FAQ

1.6.1 Compiler error: “Too many template arguments for class template \texttt{‘document\_traversal’}”

Increase \texttt{BOOST\_PARAMETER\_MAX\_ARITY} value from default 8 value. Place define like this before any Boost or SVG++ include:

```
#define BOOST\_PARAMETER\_MAX\_ARITY 15
```

1.6.2 SVG++ stops parsing on incorrect attribute value and throws exception

SVG Specification says “The document shall be rendered up to, but not including, the first element which has an error”. Therefore such behavior is correct, it will be better to fix SVG document.

But if you want to continue rendering document ignoring error in an attribute, you should inherit your class from \texttt{policy::error::raise\_exception}, override its \texttt{parse\_failed} method to not throw exception, but return \texttt{true} instead:

```cpp
struct custom_error_policy: svgpp::policy::error::raise\_exception<BaseContext>
{
  template<\class AttributeTag, \class AttributeValue>
  static bool parse\_failed(BaseContext const \&, AttributeTag, AttributeValue const \& value)
  {
    return true;
  }
};
```

where \texttt{BaseContext} is common base class for all contexts. Then pass this new class as template parameter \texttt{error\_policy} to \texttt{document\_traversal} class:

```cpp
document\_traversal<
  error\_policy<custom\_error\_policy>
  /* ... */
><::/* ... */
```
1.6.3 Compiler is out of memory or compilation takes too long

Some compilers (e.g. Visual C++, especially 2015) may have difficulties building advanced applications that uses SVG++ library. Heavy usage of templates in SVG++ itself and also in its dependencies Boost.Spirit and Boost.MPL results in large number of templates instantiated during compilation.

SVG++ 1.1 introduces some workaround about this problem - some Boost.Spirit parsers now can be moved to another translation unit at the cost of additional call to the virtual function on each value that the parser produces.

Before first SVG++ header is included some macro should be defined for each parser to be moved:

```cpp
#define SVGPP_USE_EXTERNAL_PATH_DATA_PARSER
#define SVGPP_USE_EXTERNAL_TRANSFORM_PARSER
#define SVGPP_USE_EXTERNAL_PREERVE_ASPECT_RATIO_PARSER
#define SVGPP_USE_EXTERNAL_PAINT_PARSER
#define SVGPP_USE_EXTERNAL_MISC_PARSER
#define SVGPP_USE_EXTERNAL_COLOR_PARSER
#define SVGPP_USE_EXTERNAL_LENGTH_PARSER
```

And new source file should be added to the project that contains instantiations for some templates with parameters used in the application:

```cpp
#include <svgpp/parser/external_function/parse_all_impl.hpp>

SVGPP_PARSE_PATH_DATA_IMPL(const char *, double)
SVGPP_PARSE_TRANSFORM_IMPL(const char *, double)
SVGPP_PARSE_PAINT_IMPL (const char *, color_factory_t, svgpp::factory::icc_˓
  →color::default_factory)
SVGPP_PARSE_COLOR_IMPL (const char *, color_factory_t, svgpp::factory::icc_˓
  →color::default_factory)
SVGPP_PARSE_PRESERVE_ASPECT_RATIO_IMPL(const char *)
SVGPP_PARSE_MISC_IMPL (const char *, double)
SVGPP_PARSE_CLIP_IMPL (const char *, length_factory_t)
SVGPP_PARSE_LENGTH_IMPL (const char *, length_factory_t)
```

First parameters to SVGPP_PARSE_..._IMPL() macros are type of iterators that are provided by used XML parser policy. And the other parameters are coordinate type or different factories types used.

1.7 Licensing

Library is free for commercial and non-commercial use. The library is licensed under the Boost Software License, Version 1.0.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.8 Help and Feedback

Library support and development forum

Issue tracker at GitHub

Author can be contacted at olegmax@gmail.com

1.9 Acknowledgements

Thanks to Atlassian Company for providing free Bamboo continuous integration server that made it easy to test library immediately in different environments.