Animate++ Design Documentation

Wode "Nimo" Ni - wn2155@columbia.edu
Xuanyuan Zhang - xz2580@columbia.edu

Motivation for the project

SVG stands for Scalable Vector Graphics, which is nowadays a "must" for fancy front-end in favored by both clients and front-end developers. However, the standardization of SVG has always been a big issue. Due to a lack of type-checking and other Object Oriented Programming features supported by Javascript, which have been used for over 90% of SVG libraries and editors, the syntax of SVG is expanding in a rampant manner and somehow, out of control. What we want to achieve in Animate++ is to utilize C++ features to create an easy-to-use library for both static and animated SVG in standardized manner, which can be interpreted by all major browsers. Since animation is the part with the most complicated syntax in our project, which is our major focus, we name our project as Animated++, standing for SVG Animation using C++.

Current output format

We are interacting SVG using XML file. The reason we made this decision is based on some background research on SVG and SMIL, standing for XML based SVG files.

- SVG
  - SVG stands for Scalable Vector Graphics.
  - Most used SVG libraries are all written in JS due to its tight relationship with front-end.
  - From our point of view, Object Oriented Design can largely improve the scalability and performance of a SVG library. Together with type checking and other C++ features, we find C++ a good match up for SVG manipulation and animation.
- SMIL
  - SMIL stands for XML based SVG files, the major strength of which is to store complicated SVG files together with animations. Nowadays, SMIL has been largely replaced by CSS based SVG files.
  - What charming about SMIL is its standalone feature, which is saying the whole SVG together with animation are all contained within the same file. This is not the case with CSS and Javascript, as CSS usually contains static components while Javascript is in charge of animations. Because of that, many digital artists are in favor of SMIL, and we want to save the standards of SVGs from the rampant SVG community.

With C++ and SMIL, we are about to make a change.

Source Code Explanation

The complete implementation of our codebase together with the ideas behind those are explained in manual. This is a quick view on the data structure and some important implementations we have been utilized.

Our basic class structure is shown in below:

The fundamental class of our project is Shape, which is an abstract class with high level properties of any shape object.

- `Shape`
  - private
    - attributes: A map of external attributes stored as key-value pairs.

  - public
    - animate: An animator object, containing an array of animations.

Classes inherit from `Shape` are:

- `Rect`
- `Circle`
- `Ellipse`
- `Line`
- `Polyline`
- `Polygon`
- `Path`
- `Group`

Each one containing essential attributes that are required to make it a valid shape object. There is table shown in below about default attributes, which are utilized by us when we parse SVG content from xml_node. (More details are presented in manual)

```
const DefaultAttributes default_attributes ({
    {"circle", {"cx", "cy", "r"}},
    {"rect", {"x", "y", "height", "width", "rx", "ry"}},
    {"ellipse", {"rx", "ry", "cx", "cy"}},
    {"line", {"x1", "y1", "x2", "y2"}},
    {"polyline", {"points"}},
    {"polygon", {"points"}},
    {"path", {"d"}},
    {"group", { }}
});
```

For each animation, there are a large number of public members:

- `type`
- `name`
- `attribute`
- `from`
- `to`
- `by`
- `repeat`
- `loop`
- `duration`
- `add`

With all these basic components, all kinds of animations as fancy as you can imagine performed by all different types of objects will be available.

Other than the essential data structure, we have a few more going on.

1. XML content parsing.
   We are utilizing pugixml library, which is a light-weighted general xml parser library in C++. To adapt it in fulfilling our requirements, we utilized "boost/regex" to not only search for SVG tags to be parsed but analyze each different types of shape together with their attributes being loaded into our customized data structures. The detail of which can be seen in parser.cpp and parser.hpp. Comments are well-made and we will save time for readers who are not that interested in the parser here.

2. Loading and exporting
   For loading, we either parse from XML file to extract shapes, attributes and animations. Moreover, we can create our own objects by calling the constructors of each individual object.
   For export, we support a "save" function in Shape class that can be used by all child classes of Shape, which is basically calling export_SVG for each individual child class when we output the content.

Comparison between Animate++ to some existing SVG libraries

JavaScript

Most SVG libraries are written in JS, mainly because of SVG's rising importance in front-end development. Here are a few svg libraries listed implemented in JS listed below:

- paper.js
- snap.svg
- d3.js

One of the major reasons why people favor JavaScript libraries when building SVG not only because javascript is born for front-end but because it is easy to be handed on.

What we want to propose is that what we have designed in Animate++ are equally easy to be used as those javascript libraries. Examples are given in below:

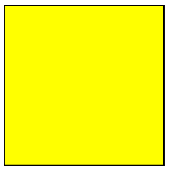
js code:

```
var canvas = SVG('canvas').size( 100,100)
rect = canvas.rect(100, 100).fill('yellow').stroke('black');
```

Animate++ code:

```
Rect r(0, 0, 100, 100);
r.attr({{"fill", "yellow"},
        {"stroke", "black"}});
```

Both code snippet brings us an SVG file of a yellow square with black stroke. SVG shown in the following.



What we are able to achieve is only to make SVG editing as simple as what has been achieved by previous works, but to utilize C++ features to make SVG manipulations more rigorous, taking advantages of type check and Object Oriented Design to bring up standardization for a more consistent SVG society.

In the nutshell, using C++ to takeover JS is how we can outplay the prior works regarding both code clarity and performance.

C++

There has not been too much prior works done in C++ as a general purpose SVG library. We found a similar one named SVG++, which is simply a SVG parser that only loads data from SVG file and help for the most common tasks on SVG.
We find Animate++ is capable of outplaying SVG++ for two major reasons.

1. SVG++ does not contain any animation. Animation is what the charm of SVG all about.
2. Even for static parts, our implementation easily outplays SVG++ on both extensibility and simplicity. An example is shown below.

Regarding a task to create a rectangle.

```
char text[] =
    TEXT(<svg xmlns="http://www.w3.org/2000/svg">)
    TEXT( <rect x="100" y="150" width="400" height="200" rx="10" ry="5"/>)
    TEXT(</svg>);

rapidxml_ns::xml_document<> doc;    // character type defaults to char
try
{
  doc.parse<0>(text);
  if (rapidxml_ns::xml_node<> * svg_element = doc.first_node("svg"))
  {
    loadSvg(svg_element);
  }
}
catch (std::exception const & e)
{
  std::cerr << "Error loading SVG: " << e.what() << std::endl;
  return 1;
}
```

What SVG++ does is to create xml code first and using parser to parse contents into its svg_element. That's their only way of initialization, not utilizing any Object Oriented Programming techniques. What we have supported is to either create objects by simply passing in parameters or loading from external svg or xml files. Our example looks like:

```
Rect r(100, 150, 400, 200, 10, 5);
```

Which is as simple as one line.

In all, we are able to provide a unique solution taking both the language advantage of C++ over Javascript and make our product as simple as javascript code.

C++17 features used

There is one C++17 feature we used in our code, fold expression of constructor, which is to let the constructor take any number of input parameters.

Taking advantage of this technique, the constructor of Group object can take any number of shape objects. We then pass them all into the ShapePtrList object, which is a vector of ShapePtrs. This technique has made our life much more convenient.

```
class Group : public Shape {
private:
    ShapePtrList shapes;
public:
    template<typename... Args>
    Group(Args&&... args)
        : Shape("group_" + std::to_string(next_id()))
    {
        (this->shapes.push_back(args.clone()), ...);
    }

    Group(ShapePtrList&);
    Group(ShapeList&);
    Group(pugi::xml_node&); // import SVG
    // void add(Shape& s, ...);
    ShapePtr clone() const { return ShapePtr(new Group(*this)); };
    pugi::xml_node export_SVG(pugi::xml_document&, bool standalone=false);
    std::ostream& print(std::ostream& out) const;
};
```

OOP design (Factory Pattern)

One OOP design technique that has been utilized in our project is factory .

According to the definition on Wikipedia, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is the scenario we encountered when we try to get shape from any given xml_node, as we cannot forecast what type of shape will be taken as our input. We therefore create a factory to load from a node, simply by create a smart pointer of class Shape and try out all fundamental shapes to find whether there is any matching. If so, we dynamically allocate an object on the address pointed by the smartptr and return the pointer. Otherwise, we throw an error.
The code is shown in below:

```
ShapePtr anipp::get_shape(pugi::xml_node node) {
    std::string name = node.name();
    Shape* res;
    try {
        if     (name == "circle") res = new Circle(node);
        else if(name == "ellipse") res = new Ellipse(node);
        else if(name == "g") res = new Group(node);
        else if(name == "circle") res = new Circle(node);
        else if(name == "rect") res = new Rect(node);
        else if(name == "ellipse") res = new Ellipse(node);
        else if(name == "line") res = new Line(node);
        else if(name == "polyline") res = new Polyline(node);
        else if(name == "polygon") res = new Polygon(node);
        else if(name == "path") res = new Path(node);
        else throw "getShape: SVG shape element tag not recognized";
    } catch(...) {
        cout << "get_shape: error when processing command: ";
        node.print(cout);
        cout << '\n';
    }
    return ShapePtr{res};
}
```