2D Animation with SVG Renderer

Elina Thadhani (elinat)
Julie Chavando (jchavand)

March 20, 2019

1 Summary

For our final project we extended Assignment 1 with support from drawing curves, lines, polygons and svg groups to support interpolation animation with keyframes. Using the rendering platform of Assignment 1 we created an interactive interpolative 2D animator that allows a user to dictate keyframes to animate and move through frames that connect them. Pixar short films have a powerful way of bringing technical feats of computer graphics through art to life. We hope to communicate the same message with our short-stop motion 2D animation and allow a user to experience building and visualizing animation.

2 Intentional Design

We began this project intending to use splines to animate 2D svgs. However, upon researching implementations of splines as polynomial interpolations, we determined that with our SVG shapes having mostly hard edges (triangles, rectangles, polygons etc), piecewise polynomial even at high degrees would not be as effective as linear interpolation. Thus we aimed to user linear interpolation to generate paths from shapes in keyframes.

In playing with user interaction with our program we decided to add the functionality of having the user advance through the interpolative frames of our animation. Thus, the user could step through frames and observe changes in SVG shapes as they animate from one keyframe to the next. We decided to handle SVG elements by calculating interpolation needed point by point such that our 2D animator could handle rotation, scaling and linear transformations of SVG elements between keyframes.

With respect to design of our keyframes, we analyzed how svg files were built (from illustrator files) and determined that if layering remained constant and only positions/sizes of SVG elements were changed from file to file, than upon parsing, the SVG elements appeared at the same index in the array of elements associated with each SVG file. Based on this observation we decided to allow for animation of entire files of SVG elements by using given keyframes to check differences in positions and sizes of SVG elements between frames. Keyframes

can additionally be ordered to any particular order based on their names in a specific folder.

3 Code Review

In order to extend our rasterizer to have 2D animation functionality, we had to expand several classes and add many individual methods.

3.1 Keyboard Event

All of our animation occurs from the keyboard event of pressing the letter "a". Pressing this key is handled in drawsvg.cpp in the char_event() method. Here we set a boolean for animation that is passed into our call to draw_svg (from software_renderer.cpp) in the method redraw(). This animation boolean allows us to render static pictures as well as animate SVGs. In utilizing the method setTab(index tab) given in the starter code to loop through key frames, we could leverage the existing implementation of redraw() and draw_svg to support our interpolative checks. We heavily edited the given setTab(index tab) method to handle switching through SVG files in a folder of SVGs to be animated. The global, interpolation_num is stored in our drawsvg.cpp file and this number can be edited to increase or decrease the number of frames the program will generate in between given keyframes.

3.2 Determining Elements to be Interpolated

The first step in interpolating between two SVG files passed into draw_svg (in software_renderer.cpp) was determining which elements were changed between the two files. Thus we loop through the array of elements for both files and compare, type by type to determine if any vertices of the elements are changed. For this we had to write a method, elemsAreDiff(SVGElement *e1, SVGElement *e2); That could take in two general element pointers, determined specific type of element (polygon, line, rectangle etc.) and called specific helper method for that type to determine if elements were different. elemsAreDiff returns a boolean that is true if elements are different and false otherwise. The index of the elements that needed to be animated (moved between the two SVG files) were stored in a vector.

3.3 Interpolation of Elements

With a vector of indices for elements that needed to be changed for a frame, we could edit appropriate elements to render intermediate frames. For each frame in between keyframes we re-rendered the entire screen based on a new array of SVG elements constructed with interpolated SVG elements. Note that we did have to clear the target buffer and re-render the entire screen for each interpolation frame. All the work for creating these new elements was completed in software_renderer.cpp.

For each SVG element to be interpolated, we looked at each vertex on the element, using our method Vector2D getCurPoint(). For each vertex we could call second helper method Vector2D calculateNewPoint() which would return the calculated new point for that specific vertex of the new SVG element. Finally, with all of the vertices of an element interpolated we could generate a new SVG element with our method SVGElement *makeNewElem();

Note that when dealing with getting vertices and creating new elements we had to handle each type of SVG differently and thus have helper methods that separately calculate new SVGs depending if the element is a line, polygon, rectangle etc.

3.4 Mathematical Justification

In order to calculate the new position for each vertex of an SVG element, we used the following equations:

$$x = x_0 + = l\sqrt{\frac{1}{1+m^2}}$$

$$y = y_0 + = m \cdot l\sqrt{\frac{1}{1+m^2}}$$

To determine the two surrounds points on the line segment and if an object was moving in the right direction. Coordinates were also calculated based on the interpolation number, such that given the number of frames between two keyframes, the new SVG elements would linearly move from one position to the next. In handling each specific point separately we allowed for animation of SVG elements with rotation, scaling and linear transformations.

4 Assets

We produced several demos and videos to display the animator we built. These are located in the folder submission (in svg/animation). We produced our own animations of abstract shapes, a model of the house from UP moving up into the sky, and finally our own take on the Pixar classic animation of the iconic lamp.
Figure 1, shows the first two self-generated frames of our Pixar and UP animations (our animator filled in the intermediary steps).

5 Installation Instructions

To properly use our 2D animator, load into the svg/animation folder a separate folder of keyframes. Note that there are certain requirements for our animator to work with a set of SVG files:

- Each SVG file must have the same number of elements.

- The elements in the SVG file must be in the same order (layering wise).
- The keyframes must be named in order such that when the names of the files are hashed, they would be sorted into the order of animation.

In order to change the interpolation number and increase or decrease the number of intermediate frames to be generated in between key frames, edit the global variable in the drawsvg.cpp and software_renderer.cpp file.
To run our code from the build folder: ./drawsvg ../svg/animation/'substitute desired folder here'. We suggest running simpler animations (such as 'triangles' or 'ball') with a smaller interpolation number to observe movement while a higher interpolation number is recommended for more complex animations like 'up' or 'pixar'.
We have also included a final_vids folder in the root directory to showcase some of our animations!

6 Acknowledgements

We would like to thank Colin especially for helping us flesh out our idea to build a 2D animator. He helped us focus on increasing the complexity of our animator rather than extend the static rasterizing elements of Assignment 1. We would also like to thank Kayvon for a great quarter in CS248, we both learned a lot and loved the class!

(a) First "Pixar" Frame

(b) Second "Pixar" Frame

(c) First "UP" Frame

(d) Last "UP" Frame

Figure 1: Self-generated keyframes