Creating 3D Foldable Papercraft from Dynamically Generated Scalable Vector

Daniel Slaughter
Grand Valley State University

Follow this and additional works at: http://scholarworks.gvsu.edu/cistechlib

Recommended Citation
http://scholarworks.gvsu.edu/cistechlib/210

This Project is brought to you for free and open access by the School of Computing and Information Systems at ScholarWorks@GVSU. It has been accepted for inclusion in Technical Library by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.
Creating 3D Foldable Papercraft from Dynamically Generated Scalable Vector Graphics

By

Daniel R. Slaughter

April, 2015
Creating 3D Foldable Papercraft from Dynamically Generated Scalable Vector Graphics

By
Daniel R. Slaughter

A project submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Information Systems

at
Grand Valley State University
April, 2015
# Table of Contents

Abstract
Introduction
Background and Related Work
Program Requirements
Implementation
Results, Evaluation, and Reflection
Conclusions and Future Work
Bibliography
Appendices
Abstract

This project demonstrates the use of Scalable Vector Graphics (SVG), Web Graphics Library (WebGL), and JavaScript to dynamically render textures onto a 3D model. The Focus is on creating a system to customize a papercraft model in a project called Paper Person. Paper Person allows designers and developers to work together to curate papercraft models, allowing builders to customize, print, and assemble them.

Introduction

The Internet has always been a place with very little output to the physical world. With the recent introduction of 3D printers, users are now able to create and export models at a consumer level. However, even the least expensive 3D printers cost hundreds of dollars, and the resin used in a moderately complex design can cost additionally tens if not hundreds of dollars. A cheaper alternative involving pre-Internet technology has been producing 3D models for hundreds of years: papercraft.

Papercraft is superset of origami where shapes are cut out of printed paper, folded, and assembled to create a 3D shape (usually humanoid). Unlike strict origami, papercraft permits cutting shapes from paper, as well as the use of glue and tape. One of the earliest forms of papercraft was paper dolls, dating back to 1810 (Figure 1) [1].

Figure 1: An early version of papercraft: the paper doll.
Papercraft models (the 2D printed page), requires a designer to illustrate and color. After the model is complete, however, there is no easy way to customize the look or color. For example, consider the paper doll in Figure 1. There is no easy way to change the dress in the lower left from red to blue.

3D models are more popular nowadays. They add depth and scale resembling real life objects. Not only is the front of the model detailed and colored, such as the clothing in Figure 1, but so too are the sides, back, top, and bottom. Take for example Figure 2 which shows an example of a model before it is assembled, and Figure 3 which shows the same model after it is assembled [2].

Figure 2: A modern day 3D version of papercraft prior to being assembled.
A 3D model gives dimension to the real life object it is meant to represent. Similar to the 2D model in Figure 1, the model in Figure 2 and Figure 3 does not have an easy way to customize the look or color. A designer has already created the shape of the model, and clothing and colors have been added to it.

What is needed is a tool designed for two audiences. For papercraft designers, a tool is needed that allows the designer to construct different shapes and textures. For example, a typical humanoid papercraft figure often has a cube for a head, but other shapes are possible. For the papercraft builder, a tool is needed to customize textures and colors for later printing. As designers create new models, they should be made available to builders as a model library. As a builder customizes a figure, it is often difficult to see the results of that customization, so the tool should display a 3D rendering of the figure in real time. At any time the builder is satisfied with their results, they can print the model.

This paper describes the design and implementation of Paper Person, satisfying the requirements mentioned above. Beyond the utility of providing a papercraft tool, the project also demonstrates the capabilities of WebGL, open source libraries that assist developers with 3D modeling, and the use of Scalable Vector Graphics (SVG) to allow for dynamic changes of colors.

Background and Related Work

Many websites exist online with pre-created static papercraft models that can be printed out. However, these websites do not allow for customization from the builder, and whatever textures and colors the model has is permanent unless the designer releases the same model with a different texture. Here are a few examples:
There are some websites with pre-created *dynamic* papercraft models. However, these websites do not come without disadvantages. Below are some examples of websites with dynamic papercraft models.

**Foldable.Me**

Foldable.Me ([https://www.foldable.me/](https://www.foldable.me/)) was a Kickstarter project funded in April of 2012 allowing a person to create a model that looks like themselves [3]. Foldable.Me requires the company print and mail the foldable model to the builder for a fee ($11.99 as of April, 2015). Foldable.Me has a good user experience while creating the model, but in an attempt to monetize its concept, lacks the ability to print or save the model locally.

![Screen capture of Foldable.Me's builder.](image)

Foldable.Me uses HTML DOM elements, positioned and skewed using CSS3 transformations, for the rotation animation and interface shown in Figure 4. The disadvantages of using CSS3 for animations is that they are very difficult to change from a developer's standpoint, and are limited in their 3D capabilities. Color choices for each texture is limited, and presumably pre-generated by a designer.
Action Cruisers

Carnival Cruise Line has a project called Action Cruisers (http://www.carnival.com/cms/fun/action-cruisers/). Action Cruisers allows a person to create a model that has Carnival branding, and is limited to select textures. Unlike Foldable.Me, Action Cruisers allows you to print the characters locally for free.

The primary disadvantage with Action Cruiser is the selection of textures is limited to Carnival Cruise Line themed attire and branding. As an example, Figure 5 shows a character in a green sports outfit branded with the team name “Cruisers”. Action Cruisers is using Flash for the 3D animations and interface. The disadvantage of using Flash is that it is not supported on modern mobile devices, nor desktops without a plugin. Color choices for each texture is limited, and presumably pre-generated by a designer.

Paper Person

In order to suit the needs of both designers and builders, Paper Person has the following features:

- It works on the Web, using existing technologies.
- It is lightweight.
- Creation of models, or using existing ones, is simple and flexible.
Designing new textures for existing models consists of the following:

- There are no limits on what textures can be designed.
- Textures are stackable, allowing for overlap.
- Textures can have any number of colors which can be dynamically changed.
- Textures are able to be created using existing technologies designers are familiar with, such as Adobe Illustrator.

The final user interface consists of the following:

- Clean and simple.
- The interface allows for selection of textures and colors.
- The interface shows the 3D representation as though it were printed.
- The interface allows the builder to rotate and view the model from multiple angles.
- A print button allows the model to be printed.

Implementation Technologies

Three technologies were primary to the implementation of the project: Scalable Vector Graphics (SVG), WebGL, and JavaScript.

Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) [4] is an image technology which is based on XML. Consider this example:

```xml
<svg version="1.1" x="0px" y="0px" width="500px" height="250px">
  <rect x="30" y="15" fill="#FF0000" stroke="#000000" width="225" height="225"/>
  <circle fill="#0000FF" stroke="#000000" cx="375" cy="125" r="100"/>
</svg>
```

We see there exists a root element `<svg>` and two child elements `<rect>` and `<circle>`, both with their own set of attributes describing their shape. When rendered, the output is shown in Figure 6.
Since the image is represented by XML, developers can programmatically update positioning and colors. For example, the blue circle in Figure 6 can be changed to green by simply updating `fill="#0000FF"` to `fill="#00FF00"`.

Modern web browsers have support for SVG, allowing them to be embedded into the Hypertext Markup Language (HTML). Embedding an `<svg>` tag can be done by including the tag into the HTML, or by drawing the graphic onto an HTML canvas element. Being able to draw SVG onto a canvas is beneficial for this project because it allows for rendering as textures onto WebGL components (see below).

Paper Person uses SVG to represent 3D objects in 2D. For example, a Paper Person’s head (a cube) starts out as a 2D representation on paper. After it is printed, it is cut out and assembled into a 3D object. A cube could have 6 sides, flaps, and fold lines as shown in Figure 7.
Although the SVG models could be created by hand using any text editor (recall that an SVG file is just an XML document), using a graphical drawing tool such as Adobe Illustrator makes the creation process much easier. A tool such as Illustrator can provide layers and sublayers, all with their own names. These names export to the SVG file as the *id* attribute for the XML element they are on. In the case of the cube example in Figure 7, the layers can be mocked up to be representative of Figure 8.

![Figure 8: Naming convention of Layers for Paper Person.](image)

The top layer “part” contains sublayers that are meant to be a grouping of parts. In this particular example, “head” is the only part. In theory there could be multiple parts here, such as “torso” or “feet”.

Within the “head” layer exists four sublayers: outline, border, foreground, background.

- The “outline” layer contains the main edging, which primarily consists of the tabs.  
- The “border” layer is very important in that it shows the fold lines, but also dictates the edges of each face of the object, which Paper Person will use to cut up and render this 2D model into 3D. The name of each sublayer within the border will dictate how a developer can reference it in the JavaScript object.  
- The “foreground” layer is where a designer would make their changes, adding any textures they wish. For the simple head part, this will be empty since no textures have been applied.
The “background” layer simply contains the fill color, which is shown as light tan in Figure 8. Naming the sublayer of background “color:skin” has importance, and will be discussed later.

Unfortunately, as CS6, Illustrator only allows for modification to the id attribute of elements (referred to as “label”), and does not allow a designer to modify other attributes. Although the ability to alter more than the id attribute is not available, the id attribute can be used by designers to describe parts of an SVG.

Web Graphics Library (WebGL)

Web Graphics Library (WebGL) [5] is the implementation of OpenGL supported by modern web browsers including Chrome, Firefox, Safari, Opera, and Internet Explorer. WebGL can render 3D objects onto an HTML canvas. Typically, developers interface with WebGL through a client side scripting language called ECMAScript, or more commonly referred to as JavaScript. JavaScript programmatically creates a 3D model to which lighting effects and textures can be added. The textures themselves can be applied based on images, existing canvas layers, or other media sources.

In Paper Person, we programmatically generate a 3D object (say a cube), map an appropriate SVG drawing onto an HTML canvas, and then apply the HTML canvas as a texture onto the 3D object (Figure 9).
Figure 9: Using SVG drawings as 3D textures.

**JavaScript**

**Libraries**

As mentioned above, Javascript is used to combine 2D SVG representations to 3D WebGL representations, as well as to provide the general user interface for Paper Person. Paper Person uses two frameworks: jQuery and Three.js.

jQuery [6] allows for easier cross-browser implementation, reducing the lines of code, than pure vanilla JavaScript would require. For example, jQuery simplifies Asynchronous JavaScript and XML (AJAX), which allows for multiple requests to a server simultaneously. Without using a framework such as jQuery, AJAX requires tedious development time and lengthy code to
support cross-browser implementation. By using AJAX effectively within the project we are able to reduce the amount of network traffic, and asynchronously preload multiple assets.

Three.js [7] is an open source framework written by Ricardo Cabello and other contributors that simplifies creating 3D objects in WebGL. By using Three.js, all of the matrix multiplication and complicated math involved with 3D graphics is abstracted by simple to use objects and functions.

Architecture

Paper Person has an Object Oriented approach to pull in the SVG, allowing for inheritance of both Model and Part for any type of model. Figure 10 shows a UML Diagram for the base classes: Object, Face, Coordinates, Layer, Model, Part. Figure 10 also shows subclasses for a humanoid shaped model: ModelHumanoid, PartArmLeft, PartArmRight, PartFeet, PartHead, PartTorso.

Figure 10: UML Diagram of Paper Person.
The object Part is a collection of Faces. In Figure 8, there are sublayers of border: top, bottom, right, left, back, front. An object PartHead can be created which has a method called getObject3D that overrides the superclass Part (see Appendix A for example source code demonstrating PartHead).

There are many references of mesh['...'] within a Part. These meshes, which are the faces within the 3D representation, are indexed automatically by their SVG identifier. In PartHead, there are references to top, bottom, right, left, back, and front which correspond to Figure 8. Once a developer creates a child class of Part, then a 3D representation can be generated (Figure 11).

![Figure 11: A 3D representation of the part “head”.

The meshes generated are the exact dimension as indicated in the SVG. This allows a developer to dynamically position the elements. Instead of the developer hardcoding the positions, the width and height of every element provided by the designer can be accessed. This has a benefit in that a designer could decide to increase the width of the head without the developer needing to recode the object. For example, instead of setting the front’s z position to 60, the following could be provided: this.face['left'].coordinates.width/2. This would move the front forward 50% of the left side’s width, keeping the head positioned around the center point of 0,0,0.

If all the parts are put together in one document, there can exist a JavaScript object representing the parts as one model. Figure 12 shows an SVG model representing a humanoid, containing a head, torso, feet, and arms.
ModelHumanoid inherits from Model and contains how the parts interact and connect to one another: PartArmLeft, PartArmRight, PartFeet, PartHead, PartTorso. Similar to Part, a method called getObject3D exists which overrides the superclass Model (see Appendix B for example source code demonstrating ModelHumanoid).

When Model.loadTemplate() is called, the SVG file (Figure 12) is loaded and a Part object is created for each part listed. Each Part is indexed by the name provided in the SVG, similar to how PartHead has a “face” indexed by what id it was given within the SVG. In Figure 8 we saw how there existed a part called “head”, which would map to our system having a JavaScript object called PartHead. In addition to head, there exists armLeft, armRight, feet, and torso.

Once ModelHumanoid is created, a developer can easily add it to an existing Three.js scene (see Appendix C for example source code demonstrating adding a Paper Person model to a Three.js scene).

The first thing a developer needs to do is set up a Three.js scene. The next step is to instantiate ModelHumanoid, and wait for the template to complete loading. Once the callback function is called, the model is loaded into the Three.js scene. To assure the model is visible in the scene, an animate function is continuously called to animate the model at approximately 60 frames per
second (fps). Once complete, a 3D model of ModelHumanoid, composed of its parts, will be visible (Figure 13).

![Figure 13: A 3D representation of ModelHumanoid.](image)

To apply a texture to head, such as hair, a designer edits the same Scalable Vector Graphic (SVG) shown in Figure 7 and Figure 8. Similar files for the other parts also exist. When a designer edits the SVG, they add any design or texture they wish to the foreground layer. If there is a color the designer would like the developer to be able to change, they can indicate that by giving the layer with the color an *id* of color:1, color:2, etc. Similarly, if the color should match the skin color they can provide an *id* of color:skin. Figure 14 shows what a hair style may look like on head, with one color option.
By default when a texture is loaded into the JavaScript object Layer, the default colors are used (Figure 15). Multiple layers can be added to the same part, giving developers the ability to add eyes, hats, or any texture a designer creates.

To add a layer, such as the hair in Figure 14, the Model needs to first be loaded. There are two options when loading a layer. One option is to instantiate the Layer object within the loadTemplate callback of ModelHumanoid. The other option is to preload all of the Model and Layer objects and their templates via AJAX. Once the Layer object successfully loads it is added to the ModelHumanoid’s PartHead. After the next `redraw()` finishes, the model will look like Figure 15.
It is possible to change any color indicated by the designer (See Appendix D for example source code demonstrating how to load a layer and change its color).

All colors exist in an array on the Layer object. If a designer marks as layer with an id of color:1, then the color object on Layer will have an item indexed at 1. Each color is an object with two properties: hex, original. The hex property allows a developer to update the color as done in Appendix D (Figure 16). The original property contains the default color provided by the designer.
loaded, the Layers can be added to the Parts. At any time colors on a Layer can be programmatically changed.

**Builder Interface**

Giving these capabilities, we are able to create an interface for a builder to use (Figure 17). This interface is clean and simple, allows for selection of textures and colors, shows the 3D representation as though it were printed, allows the user to rotate and view the model by clicking and dragging, and has a print button which allows the model to be printed.

![Figure 17: The interface of Paper Person](image)

Once the builder is satisfied with their model, they can click the “Print” button which will flatten the model back into 2D in its original form from Figure 12, but with all of the textures applied. Figure 18 shows an example of a model when printed. Notice how the model has hair, eyes, a dress, and shoes layer applied, which were not in the original form.
After printing, cutting, folding and gluing the model together, the builder will have a physical 3D papercraft (Figure 19). This model allows the head, torso, and feet to detach from one another by slits and flaps. If multiple Paper Persons were printed, their parts could be interchanged to allow the builder to swap out heads, torsos, and feet from one model to the next.
Results and Evaluation

Although a great deal of time was spent working on this project, there could be room for improvement. The design knowledge of Adobe Illustrator is limited, and working with a designer to discover how they use the product could be beneficial for future iterations of Paper Person.

Even with knowledge of how a designer uses Adobe Illustrator, the ability to critically look at a design is something this project needs. From a developer’s standpoint, it functions well, but an improved user interface and library of textures could be provided.

If designers could easily upload and share SVG files with builders, it would provide an opportunity for a growing library of textures. Adding this feature should not be too complicated. A page to upload and manage a designer’s entries, and an admin toolset to control which entries were uploaded, would be needed.

A developer is currently needed to position the faces of the parts, and the parts within the model. It may be possible for the designer to indicate in the SVG how the faces and parts are rotated and positioned. This would be simple if a future version of Adobe Illustrator allowed designers to edit more than the \( \text{id} \) attribute. Paper Person could also be reworked to accept a delimited list of arguments through the \( \text{id} \) attribute. This would allow a designer to give indication to how each item is positioned in its 3D space.

When starting this project there was not any familiarity with Three.js or 3D programming. As development on this project progressed, many notes were taken to improve the performance and interface of Paper Person, but due to a lack of time those performances were not made.
Going back and refactoring the code to include performances and simplifications is ideal for future development.

Currently the only way a face of a part can be used in Paper Person is if it is a `<poly>` element with the `points` attribute. If an element is a `<rect>`, which just has an `x, y, width, and height` attribute, it is not supported. Code will be added to Paper Person so any SVG shape can be used.

Cross browser support is lacking with WebGL. To accommodate for builders using older web browsers, fallbacks would be needed. These fallbacks would allow the 3D model to be represented on a 2D canvas, or as flat images. Providing these fallbacks may be beneficial to the adoption of this project to both designers and builders.

**Reflection**

Overall this project provided a great deal of learning experiences. Expanding knowledge into the 3D realm has always been something of interest, and has provided much joy. Learning more about 3D, WebGL especially, can be done by continued efforts on this project. As mentioned above, there are many opportunities for this project. The overall goal for future iterations of Paper Person will be to reach a broad audience of designers and builders.

**Conclusion**

Paper People illustrates what can be done using modern Web technologies. Since the inception of the Web, support for graphics have steadily increased. Web standards have evolved to provide sophisticated support for graphics, and the SVG standard allows developers to programatically interrogate an image without resorting to bit manipulation. Web browsers have matured to support 3D modeling, and by using appropriate libraries, is not overly complicated. Web standards have certainly evolved to allow the browser to be a development platform on par with the desktop.
Bibliography


Appendices

Appendix A

PAPERPERSON.PartHead = function (backgroundColor) {
    PAPERPERSON.Part.call(this, [backgroundColor]);
    this.type = 'PartHead';
};
PAPERPERSON.PartHead.prototype = Object.create(PAPERPERSON.Part.prototype);
PAPERPERSON.PartHead.prototype.constructor = PAPERPERSON.PartHead;
PAPERPERSON.PartHead.prototype.getObject3D = function () {
    if (this.obj3d == null) {
        this.obj3d = new THREE.Object3D();
        var mesh = {};
        for (var faceIndex in this.face) {
            mesh[faceIndex] = this.face[faceIndex].getMesh();
            this.obj3d.add(mesh[faceIndex]);
        }
        mesh['front'].rotation.set(0, 0, 0);
        mesh['front'].position.set(0, 0, 60);
        mesh['back'].rotation.set(0, this.degreeToRadians(180),
            this.degreeToRadians(180));
        mesh['back'].position.set(0, 0, -60);
        mesh['left'].rotation.set(0, this.degreeToRadians(90), 0);
        mesh['left'].position.set(60, 0, 0);
        mesh['right'].rotation.set(0, this.degreeToRadians(-90),
            0);
        mesh['right'].position.set(60, 0, 0);
        mesh['bottom'].rotation.set(this.degreeToRadians(90), 0, 0);
        mesh['bottom'].position.set(0, -45, 0);
        mesh['top'].rotation.set(this.degreeToRadians(-90), 0, 0);
        mesh['top'].position.set(0, 45, 0);
    }
    return this.obj3d;
};
PAPERPERSON.PartHead.prototype.clone = function (obj) {
    if (obj === undefined) { obj = new PAPERPERSON.PartHead(); }
    PAPERPERSON.Part.prototype.clone.call(this, obj);
Appendix B

PAPERPERSON.ModelHumanoid = function () {
    PAPERPERSON.Model.call(this);
    this.type = 'ModelHumanoid';
    this.setTemplate('/files/paperperson/model/humanoid.svg');
    this.animation = {
        headBounceDirection: -1,
        headBounceOffset: 0,
        headTilt: 1,
        pixelsPerSecond: 1
    };
    this.armLeftPivot = null;
    this.armRightPivot = null;
};

PAPERPERSON.ModelHumanoid.prototype = Object.create(PAPERPERSON.Model.prototype);
PAPERPERSON.ModelHumanoid.prototype.constructor = PAPERPERSON.ModelHumanoid;
PAPERPERSON.ModelHumanoid.prototype.animate = function (timeDiff) {
    if (this.part !== undefined) {
        if (this.animation.headBounceOffset >= 0) {
            this.animation.headBounceDirection = -1;
        } else if (this.animation.headBounceOffset <= -30) {
            this.animation.headBounceDirection = 1;
        }
        this.animation.headBounceOffset +=
            this.animation.headBounceDirection * timeDiff *
            this.animation.pixelsPerSecond / 1000;
        if (this.part['head'] !== undefined) {
            var head = this.part['head'].getObject3D();
            head.rotation.x =
                this.degreeToRadians(Math.sin(this.animation.headBounceOffset)*3);
        }
        if (this.armLeftPivot !== undefined) {
            this.armLeftPivot.rotation.x =
                this.degreeToRadians(Math.sin(this.animation.headBounceOffset)*25);
        }
        if (this.armRightPivot !== undefined) {
            return obj;
    });
this.armRightPivot.rotation.x =
this.degreeToRadians(Math.sin(this.animation.headBounceOffset)*25);
} }
PAPERPERSON.ModelHumanoid.prototype.getObject3D = function () {
    if (this.obj3d != null) {
        return this.obj3d;
    }
    var rotate90 = this.degreeToRadians(90);
    this.obj3d = new THREE.Object3D();
    for (var partIndex in this.part) {
        var partObject = this.part[partIndex].getObject3D();
        if (partIndex == 'head') {
            partObject.position.set(0, 102, 0);
        } else if (partIndex == 'feet') {
            partObject.position.set(0, -88, 20);
        } else if (partIndex == 'armLeft') {
            this.armLeftPivot = new THREE.Object3D();
            this.armLeftPivot.add(partObject);
            this.armLeftPivot.position.set(50, 45, 0);
            partObject.rotation.set(rotate90,
            this.degreeToRadians(23), -rotate90);
            partObject.position.set(2, -45, 0);
        } else if (partIndex == 'armRight') {
            this.armRightPivot = new THREE.Object3D();
            this.armRightPivot.add(partObject);
            this.armRightPivot.position.set(-50, 45, 0);
            partObject.rotation.set(rotate90,
            this.degreeToRadians(-23), rotate90);
            partObject.position.set(-2, -45, 0);
        }
    }
    if (partIndex != 'armLeft' && partIndex != 'armRight') {
        this.obj3d.add(partObject);
    }
}
return this.obj3d;
};
PAPERPERSON.ModelHumanoid.prototype.clone = function () {
    var obj = new PAPERPERSON.ModelHumanoid();
PAPERPERSON.Model.prototype.clone.call(this, obj);
    return obj;
};

Appendix C

// Set up a very basic Three.js WebGL component
var canvasWidth = 800;
var canvasHeight = 1200;
var renderer = new THREE.WebGLRenderer();
renderer.setSize(800, 1200);
renderer.setClearColor(0xffffffff);
document.body.appendChild(renderer.domElement);
var camera = new THREE.PerspectiveCamera(50,
    canvasWidth/canvasHeight, 1, 2500);
camera.position.z = 450;
var scene = new THREE.Scene();
scene.add(new THREE.HemisphereLight(0xffffffff, 0xe0e0e0, 1));

// Create and add ModelHumanoid to the scene
var humanoid = new PAPERPERSON.ModelHumanoid();
humanoid.loadTemplate(function() {
    scene.add(this.getObject3D());
});

// Continuously render any changes to the scene
var lastTime = 0;
animate();
function animate() {
    var time = (new Date()).getTime();
    var timeDiff = time - lastTime;

    // Animate the humanoid
    humanoid.animate(lastTime);
    lastTime = time;

    // Update the scene
    renderer.render(scene, camera);
    requestAnimationFrame(animate);
}
Appendix D

// Set up a very basic Three.js WebGL component
var canvasWidth = 800;
var canvasHeight = 1200;
var renderer = new THREE.WebGLRenderer();
renderer.setSize(800, 1200);
renderer.setClearColor(0xffffff);
document.body.appendChild(renderer.domElement);
var camera = new THREE.PerspectiveCamera(50,
canvasWidth/canvasHeight, 1, 2500);
camera.position.z = 450;
var scene = new THREE.Scene();
scene.add(new THREE.HemisphereLight(0xffffff, 0xe0e0e0, 1));

// Create and add ModelHumanoid to the scene
var humanoid = new PAPERPERSON.ModelHumanoid();
var layer = new PAPERPERSON.Layer('hair.svg');
humanoid.loadTemplate(function() {
    scene.add(this.getObject3D());
    layer.loadTemplate(function() {
        this.obj.color['1'].hex = '#0000FF';
        humanoid.part.head.addLayer(this);
    });
});

// Continuously render any changes to the scene
var lastTime = 0;
animate();
function animate() {
    var time = (new Date()).getTime();
    var timeDiff = time - lastTime;
    humanoid.animate(lastTime); // Animate the humanoid
    renderer.render(scene, camera); // Update the scene
    lastTime = time;
    requestAnimationFrame(animate);
}