**Python FBX-Parser - Tools and Middleware Assignment 2**

This is the write-up for the Tools and Middleware second coursework assignment.  The project was to create an FBX parser that would read a directory full of FBX files and create corresponding SVG files which are displayed as thumbnails on a webpage.  The group for this project consisted of Mircea Catana, Jean-Pascal Evette and Adam Joyce.  We chose to implement the project a number of different ways.  Due to having several implementations, throughout the write-up I will not reference specific functions, but instead focus on the different steps required to achieve the result.

**Method 1: SVG with ECMAScript**

This section focuses on the method of generating the SVG files with an ECMAScript to perform all the required actions on the models being drawn.

We begin by using the ‘Python FBX’ binding to parse each of the FBX files found in a directory.  For each file an FBX model was selected.  We obtain the mesh face data for each model, storing it as a string ready to be written into the ECMAScript generated in the SVG file.  The data is stored as an array with each element being another array of four vertex control point indices.  An empty depth array with a size corresponding to the number of faces in the model is also generated.  This is to be used to help determine the different depths of each face in the model and thus the order in which they should be drawn.

To use these face vertex control point indices to both draw and manipulate the model we also need to store the x, y and z coordinates of the model.  Each of these are stored in a separate array, using their smallest values as an offset to ensure all coordinate values are positive.  This results in the entire model being displayed within the viewable bounds of the page.  The central x, y, and z coordinates are also taken so that the models can be correctly rotated to a desired position.

The remaining code written into the ECMAScript is a series of functions that manipulate how the model is drawn and viewed.  We begin by rotating the model’s coordinates into an appropriate position around the required axes.  Another of these functions stretches the model to fit the full SVG window.

Before our SVG model image can be constructed, we must determine the order in which the faces should be drawn.  We do this by sorting the faces by their depth, and drawing the closest faces first.  Initially, we compute a mean depth value for each face by summing the z coordinates for each vertex in each face and dividing by the number of vertices.  These values are stored in an array and used to determine the closest faces from the viewing angle.  The closest face’s face array indices are stored first in the depth array we wrote into the script earlier.  This way when building the attributes of the SVG path elements to draw each face, we are able to use the index values stored in the depth array to identify which face stored in the face array we should be dealing with first.

The colouring of each face on the model is dealt with in our Python script when we construct the string of path elements to be written in the SVG file.  Since we are storing the faces by depth in the ECMAScript, we are in need of one path per face to differentiate their colours from one another.  We construct an RGB value for each face by slightly altering the original colour of the FBX model.  These values are used to define both the stroke and fill attributes of each of the path elements.  This produces a simplistic shading effect that differentiates each face from one another when the image is viewed.

To conclude the generation of our SVG file we take the concatenation of each of the above mentioned content strings and write it into an appropriately named and located SVG file.

The web-page is served up using a simple Python webserver which takes the FBX file paths and generates an SVG file for each.  These SVG file images are formatted into small thumbnail-sized images that can be viewed when accessing the server.

This application can also make use of a GitHub webhook to upload new pictures into the project. The webhook is designed to constantly monitor for new commits.  Whenever a commit is created, the webhook creates a POST request which is sent to the webpage.  This request contains information about the commit stored as a JSON object. In addition to the do\_GET function, a do\_POST function is implemented to handle the incoming requests from the webhook.  This function catches the comment from the latest commit and ensures that it conforms with the expected format for uploading new images.  If no problem arises, the FBX file linked in the comment will be downloaded and placed into the appropriate FBX folder.  The next time a user opens the page, the corresponding new SVG file will be created and its thumbnail will be displayed.

**Method 2: Mircea writes stuff in this bit!**

**References**

The main reference source materials that we used for this project is listed below:

<http://www.petercollingridge.co.uk/blog/rotating-3d-svg-cube>

<https://developer.github.com/webhooks/>

**Repositories**

Jean-Pascal Evette (SVG with ECMAScript):

Mircea Catana (Mircea’s awesome method):

Adam Joyce (SVG with ECMAScript):