Converting 2D Shapes into Cuttable Furniture Boards

TRISTAN SCHULER
George Mason University

1. Introduction

This software is the second step in creating 3D tangible furniture from a 2-D sketch. The initial processing converts a 2D vector image into a simple 3D model. The data is then exported in a JSON format with information about the furniture including the item's name, as well as a list of boards and joints. This second stage imports the JSON file of furniture, manipulates the board shapes, and finally exports a new, updated JSON file along with complimentary SVG images for each board. The software is written in java with JSON parsing provided by Google's gson library. The program can be run by launching the UpdateBoards.java file.

2. JSON File Structure

The software requires a JSON file in order to run. The structure of the file must match the structure define by this software when deserializing the data. Currently the program reads .txt files in JSON format rather than a .json file.

Each JSON file must include one furniture object, which all other data is encapsulated within. Each furniture item includes the name of that item, as well as a list of boards and a list of joints. The lists of boards and joints contain singular board and joint objects, respectively. All necessary joint and board information should be defined in the JSON file before running it with the software. A key assumption that is currently made in the software is that the JSON data is correct and all joints have corresponding boards in the file with joint connecting lines within board coordinates. JSON data should be analyzed for correctness in stage one before being sent to this program.

A board object contains the board name, board thickness, main coordinates, and a list of holes. The main coordinates list is an array of x & y-coordinates. The list of holes is similar to the main coordinates list, but instead is a two-dimensional list of coordinates, with each inner list corresponding to one hole. The holes list is initially empty, and could remain empty depending on the joint category.

The joint objects include a plug name, receptor name, joint category, plug connecting line, and receptor connecting line. Each joint will always only include 2 boards: a plug and a receptor. Plugs are the main board, and receptors are the secondary board. Depending on which joint the user selects, plugs and receptors can switch identities (such as with edge joints). The plug and receptor connecting lines are 2 dimensional arrays that include the x & y-coordinates for the starting and ending point of the joint line. The connecting lines should either be on the edge or somewhere on the inside of the shape, defined by the main coordinates of a board. Joint connecting lines and holes should always be given in counterclockwise orientation, whereas main coordinates should be clockwise.

JSON files can be created for testing with the CreateJSON.java file. This file does not check for correctness of the data, so when manually entering data with this file, the user must submit accurate data that can be properly manipulated. CreateJSON.java also creates initial SVG files for each board to examine before and after shapes. DeserializeJSON.java reads input JSON data and creates a corresponding furniture object.

3. Software Overview

The software includes eight files and eleven java classes. Only two of the classes are executable: CreateJSON.java, and UpdateBoards.java. See A-1 for a flow chart showing how the files call each other.

CreateJSON* is for creating testable JSON data. The file also includes the furniture, board and joint classes. The furniture class is the main object created from a JSON file. The user can manually input the information within the code to test different types of joints on an item of furniture. Follow the table example currently included with the code to test a different item of furniture.

DesializeJSON* is for reading JSON data and converting the information into a furniture object. The JSON data must be of the same structure defined by the CreateJSON file, otherwise bugs will arise.

*Both CreateJSON and Deserialize JSON include google-gson. Gson is a Java library that can be used to convert Java objects into their JSON representation as well as convert a JSON string into an equivalent java object. [1] The gson library is included in all files that involve parsing and creating JSON data.

UpdateBoards is the workhorse, executable file that calls all of the other classes. UpdateBoards first deserializes JSON data, asks users for a sequence of joint specifications, then exports a new JSON file with the updated board shapes.

UpdateShapes is called by Update boards to create a list of coordinates that will be used by UpdateBoards. The file either creates a list of holes, or the teeth coordinates for plugs. This information is sent back to UpdateBoards where each board object is updated.

Plugs is called by UpdateShapes. Plugs currently creates two different types of plugs, standard square tooth plugs for all joint types, and the negative square tooth plugs to compliment standard plugs.

Receptors is also called by UpdateShapes. Receptors determines the size holes needed to compliment the square tooth plugs and returns a list of hole coordinates.

[1] "google-gson." Github.com. https://github.com/google/gson

Rectangles is called by Receptors. The class creates a 2-dimensional list of doubles, with each inner list representing a single rectangular hole.

ExportSVG analyzes the main coordinates of each board and exports unique SVG files named using the board names.

4. Software Functionality

UpdateBoards in the main source for processing new shapes. First, the user must type in a path of their JSON file, which currently requires a .txt extension. DeserializeJSON is then called to create a new furniture item. ExportSVG is also called to create initial SVG files before they are updated.

Next, the determineJointType method is called for the furniture item. This method determines if a joint is an edge joint or an internal joint by checking if the lines are coinciding and then sets jointCategory appropriately.

determineJointType works by calculating the slope of a joint connecting line and the slope of each line on the main coordinates of a board. If the two slopes are different, the method returns false and moves onto the next joint. If they are equal, the helper method checkCoincidence is called. checkCoincidence checks if the connecting lines are within the min and max coordinates of a particular line on the board's main coordinates. If true is ever returned, jointType is set to edge, and if not, jointType is set to notEdge.

Next, each joint is printed to the user one at a time by calling the Joint toString method. The program then asks the user for the following information from the user:

1. Type of Joint
2. How many teeth for the joint
3. Board Thickness

When the user is asked for their desired type of joint, different menus are given to the user based off of the jointCategory of each particular joint. Currently the program handles one joint type for each category; Squaretooth for notEdge and Edge_Squaretooth for edge.

UpdateShapes is called by updateBoards and the squareToothSpacing method is run. squaretoothSpacing determines the vector representation of the connecting line and stores the vector in a size 2 array. The angle and distance are both required for all methods in the plug and receptor classes. The direction is required for receptors.

The connecting line angle is determined with the following equation:

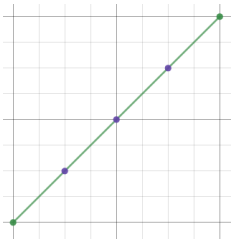
$$\Theta = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

A check is made that sets the angle to π if the x-coordinates are equal.

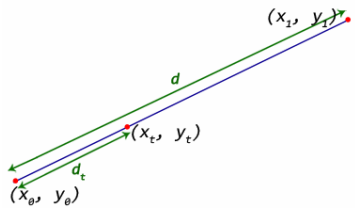
Internal Receptor Joints

For internal joints (plug and receptor type joints), the main board has holes in it and a second board seamlessly fits together. The size of the holes is determined by the tooth size and thickness of the plug board. There will always be an equal amount of receptor holes to plug teeth.

The algorithm for determining receptor joints determines the center points along a connecting line. All teeth and holes have an equal distance between a tooth/hole and the space between each tooth/hole. For instance, a receptor joint that has three teeth will have three center points along the connecting line.



The center points are calculated using a ratio of distances to determine each new x and y-coordinate.



The section distance is the total distance of a connecting line divided by the number of section points. There are 2*(numberOfTeeth*2)-1) section points for any given connecting line. Each center point is determined with the following equation:

$$(x_t, y_t) = (\big((1-t)x_0 + tx_1\big), \big((1-t)y_0 + ty_1\big),$$

where. $t = \frac{dt}{total\ distance}$

dt is defined as the section distance*section point number. Therefore, each time a center point is created, dt increases until all center points have been made.

Once all center points are determined, the holes are made by calling the rectangle class. For now, the program only makes rectangular square tooth receptor holes. The rectangle coordinates are also given in CCW orientation and all of the new points are determined with vectors to ensure rectangles can be made around a center point at any angle.

A rectangle is made around a center point with the x-direction corresponding to the section distance*2 and the y-direction being the plug's board thickness by calling the createReceptorRectangleCorners method. Next, the createReceptorRectangleCorners method is

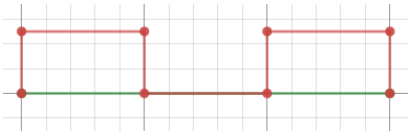
called to rotate the points properly. This method uses the following equation to rotate each point by the angle passed into the rectangles class:

$$(x', y') = ((xcos\theta - ysin\theta), (ycos\theta - xsin\theta))$$

The list of rectangles is then passed back to UpdateBoards and set to the holes variable of the appropriate board.

Plugs

Plugs are determined in a similar fashion to receptors where the teeth are defined by the section distance. Each tooth is defined by creating 4 new points.



If the bottom right corner of the tooth is not equal to the endpoint of the connecting line, a new point is made section-distance*2 units away and another tooth is made until the endpoint is reached. The tooth height is defined by the receptor board's thickness and the length is defined as section-distance*2; the space between each tooth is also section-distance*2.

The list of plug coordinates is then passed back to updateBoards and the main coordinates of the appropriate board are altered to include all of the new plug coordinates along the connecting line.

Edge Plugs

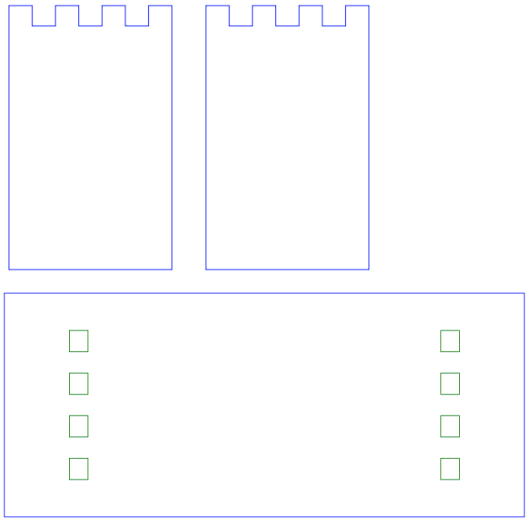
Edge plugs are very similar. The main board is specified by the user and made the same way mentioned above. The secondary board then must have a negative version of the teeth to compliment the main board. So, instead of first extruding a tooth outward, a point is made section-distance*2 away before any teeth are created. This information is then returned to updateBoards.

Plugs and receptors give the user the option to be hidden joints. This means that the board would not be cut all the way through, to hide the secondary board. The user is prompted to enter a percentage of how much of the joint should be hidden and the plug thickness parameter is updated

4. Outcomes

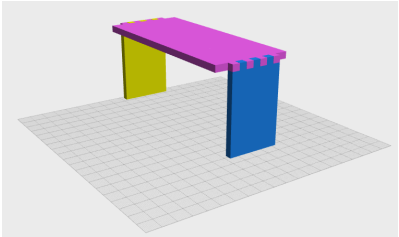
After updateBoards has been run, exportSVG is called to make a visual representation of each updated board. The exportSVG class works by plotting each point in the main coordinates as a path. Each hole on a receptor board is added as another path.

Final updated boards for a table with 4 teeth for each joint and a board thickness of 25 units would output the following SVG images:



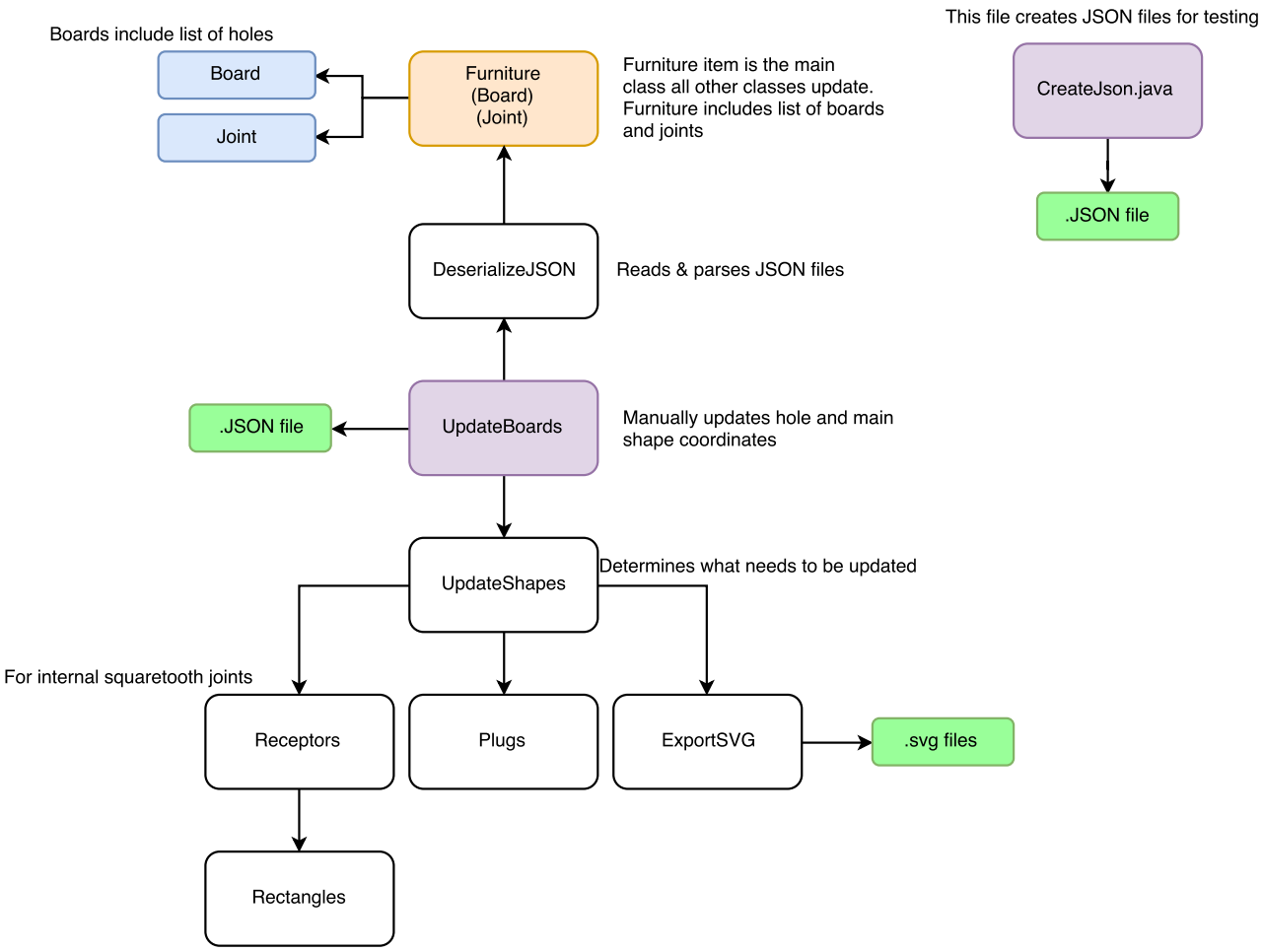
Future updates to this software will include a larger menu of joint types to select from. Exceptions should also be added to the program. As of now, several assumptions are made about the correctness of inputted data.

Another possible addition to this software will be adding JSCAD support. JSCAD is a browser version of SCAD and can produce 3D representations of objects. The following model table was made by using the program to create edge joints with 4 teeth on each side:



One drawback to JSCAD is that it uses a different distance unit than exportSVG. The SVG board thickness is 25, but the extrusion height in JSCAD for each board is 7. The two legs were also manually rotated and translated to fit with the table top rather than fitting together automatically.

The final step in converting the 2D sketches into 3D models is to cut the boards out and assemble the pieces. This will most likely be done using a CNC machine or laser printer. Therefore G-code support will eventually need to be added for the third stage of this project.


Boards include list of holes
Board
Joint
Furniture
(Board)
(Joint)
Furniture item is the main class all other classes update. Furniture includes list of boards and joints
This file creates JSON files for testing
CreateJson.java
.JSON file
DeserializeJSON
Reads & parses JSON files
.JSON file
UpdateBoards
Manually updates hole and main shape coordinates
UpdateShapes
Determines what needs to be updated
For internal squaretooth joints
Receptors
Plugs
ExportSVG
.svg files
Rectangles