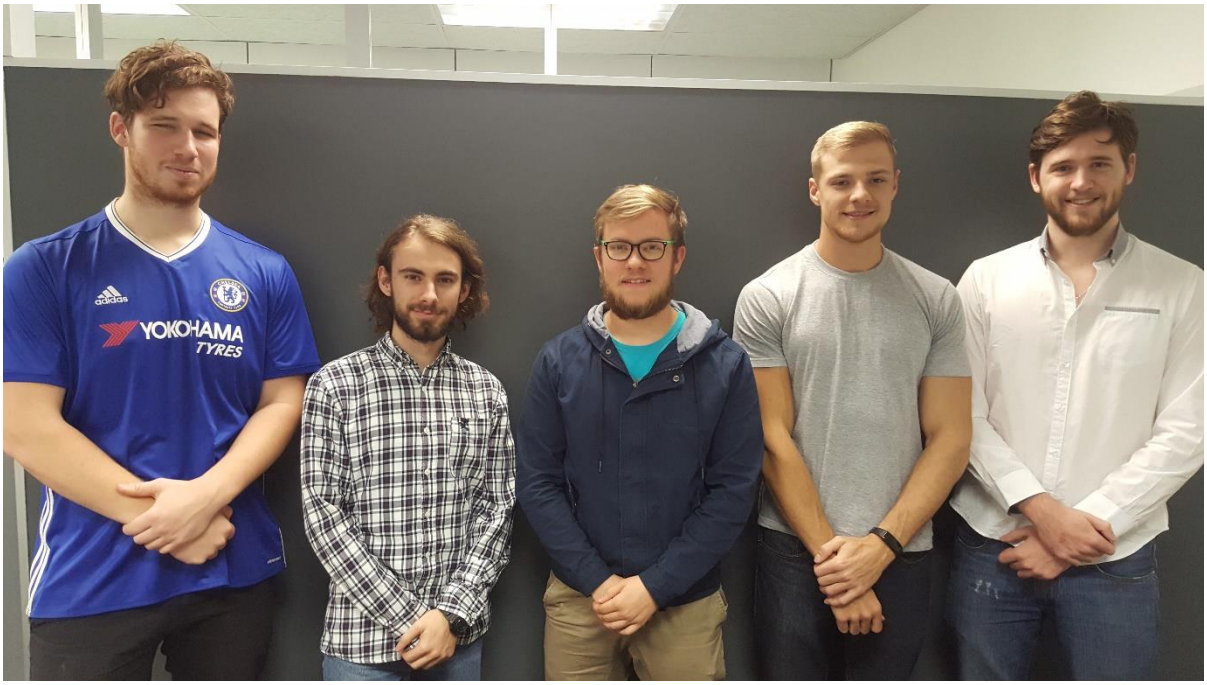


Absolute|0|

Voxc.js:
User manual

Name	Student Number
Chris Dreyer	15072623
HD Haasbroek	15046657
Cameron Trivella	14070970
Pearce Jackson	14044342
Idrian van der Westhuizen	15078729

Contents

System overview:

Voxc.js is meant to be a lightweight, easy to use library for converting voxel objects into JavaScript matrixes for easy manipulation. These matrixes can then be used for creating more meshes that have different goals.

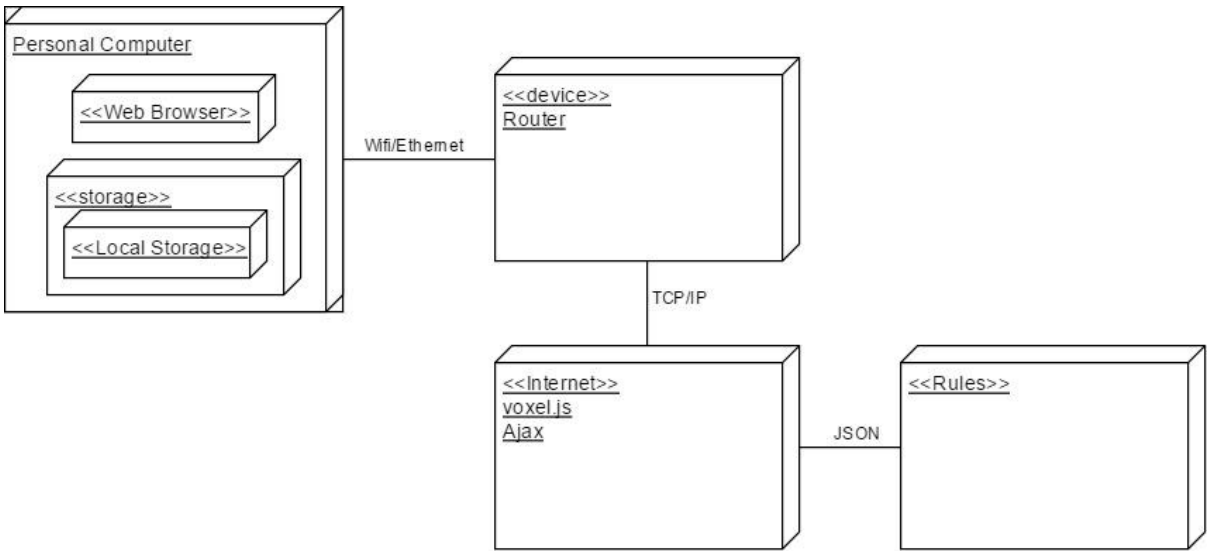
This library will contain a series of converters that all have a different purpose. The creation and formatting of these converters stay consistent by using JavaScript matrixes as input and output.

System Configuration:

The voxc.js library is a set of node modules that were all bundled together into a single JavaScript file with Browserify, in order for it to be easily added and used in different webpages via a script tag.

All the modules and classes were coded using typescript in order to make debugging easier during development.

All object models that were used and tested were created by MagicaVoxel.



Installation:

voxc.js is used by downloading the standard or minified JavaScript file and including it in your project.

The library and all of its source files can be downloaded from our GitHub page:

https://github.com/Idrian/Absolute-0-

If you wish to continue development please be sure to download one of the branches and then from your node.js command line type “npm install” once in the branch directory, this will add all the needed node modules for development onto your PC. To compile the library again type “browserify build/main -p [tsify] -s ‘module_name’ > ‘librarName.js’ “, in the node

command line, additionally node_module and libraryName.js can be what you choose, you should just remember to then add your module or library accordingly to your webpage.

Getting started and Using the System:

If you have not downloaded the voxc.js library or compiled your own version see "Installation" before continuing.

After you have downloaded or compiled the library you can proceed to add it to your webpage via a script tag as follows:

```
<script src="voxc.js" type="text/javascript" charset="utf-8"></script>
```

The Canvas and the scene:

Since all the models are Three-js models you will need a canvas and a scene to display or at least visually work with the outputted models (you don't have to be able to see them in order for the converters to work). First things first you can choose to create your own Three-js scene or you can use our voxcJSCanvas class made to ease creation of the scene and the display of the models.

If you decided to create your own scene you can continue to the actual converters section below this.

If you are still here that means you're either lost or want to use our voxcJSCanvas class, we'll assume the latter for now. To use our class simply create a div element in your html page and give it an appropriate id:

```
<div id="voxelDemo "></div>
```

Then in your javascript or typescript file or even the html page itself via a script tag you simply construct our VoxcJSCanvas class as follows:

```
let demo_canvas = new voxJSCanvas("voxelDemo");
```

voxJSCanvas attributes and methods:

Attributes:

```
public scene : THREE.Scene;
private renderer : THREE.WebGLRenderer;
private voxel : THREE.Group;
private controls : OrbitControls;
private ambientLight : THREE.AmbientLight;
private light : THREE.PointLight;
public camera : THREE.PerspectiveCamera;
```

- Only the camera is public to make changing and attaching controllers to it easier, do not alter it without knowing what you are doing.

Methods:

```
constructor(containerID : string)
```

- This methods construct the voxcJSCanvas class and is called with new voxcJSCanvas().

```
public CameraPosition(x : number,y : number,z : number)
```

- This is to change the x,y and z postions of the camera.

- `public setDimensions(width : number, height : number)`
  - Sets the width and height of the scene, if not set the default is 800 by 400.
- `public setMesh(inputMesh : THREE.Group)`
  - This is used to change the mesh/model being displayed. Only one mesh can be displayed at a time using this class, if you wish to view more than one at a time you must create your own Three-js scene and simply add the models.
- `public setGridHelper(x : number, y : number, z : number)`
  - This will display a grid at the bottom of the model. This method never needs to be called but could help with knowing the exact dimensions of your model.
- `public setBackgroundColor(color:number)`
  - Uses a hex number value (0xFFFFFF = white) to change the background of the scene, if not set default is white.
- `render()`
  - Used to render the scene at a stable framerate, cannot be called or altered by the user.
- `start()`
  - Used to start the render method, needs to be called in order for the rendering to, well start.

OBJ conversion a.k.a converter one:

This class is the class that takes in a .obj file created with MagicaVoxel and converts it into a 3d string array.

How it works (only for future developers and curious folk):

The program begins by accepting a .obj voxel file uploaded by the user. It then reads through the file line by line, looking for the following at the beginning of each line: “vn”, “vt”, “v” or “f”. It ignores the other lines. It sorts the usable lines in the following manner:

- “vn” lines are added to an array of vertex normals (they indicate the outward facing side of the face)
- “vt” lines are added to the texture coordinate array (these indicate the position of the texture/color in the .png associated with the .obj file)
- “v” lines are added to the vertex array (these contain the x/y/z position of the vertex they represent). It also compares these to the current largest and smallest values for each, to determine the eventual maximum and minimum values of the object.
- “f” lines contain the per-face information and are sent to the “setMatrix” function, which will be explained later (these lines contain code for the normal, texture coordinate and three coordinates associated with the face they represent)

“setMatrix” extracts the values of the normal, texture coordinate and the three vertex coordinates from the line and uses them, in conjunction with the aforementioned arrays, creates a new “face” object, containing all the pertinent information about each face. It then normalizes the coordinates by rounding

them to the nearest whole number and adding the lowest x, y or z value (depending on which axis they belong to), to them to ensure the lowest possible coordinate is 0 while

maintaining the correct orientation with the other blocks. It adds this "face" object, with the normalized values to the array of "face" objects.

From there these objects are sent to the "addCoords" function, where the normal are used to deduce which two dimensions are being altered in the face. It then compares the relevant coordinates of each of the three vertexes to determine which is the right angle (which will be used later). After it has set this information, it uses it to create an "angle code", indicative of the right angle of the face e.g. an increase in the x axis and a decrease in the z axis result in a code of +/0/-.

From here it enters "buildMatrix", where the final, three-dimensional matrix is created using the max values of each dimension. "buildMatrix" sends the faces one at a time into the function "add". "add" is the crux of this section of the code. Using the normal to deduce the two altering dimensions, it uses the two angles, other than the right angle, to calculate the values of gradient and intercept in the straight-line formula y = mx + c (or y = mz + c or z = mx + c). It then uses the resulting formula to calculate the location of the coordinate of the slope of the hypotenuse for each value of the corresponding, alternate side. Using the angle code to infer the orientation of the face, it determines whether the point calculated on the hypotenuse is the starting point or end point of the next function "putIn" which cycles through each level of the face, adding each block from the point of origin to the end point (each being either the point on the side of the face or the point calculated on the hypotenuse). Once this process is complete, the final matrix is ready to be transferred.

Using it:

As of this moment there is only one method needed to be known to the user and used by them, that is the constructor. The constructor takes in the File object to be processed as well as a callback method, this callback method is created by the user and is meant to be a what need to be done once the converter is done loading the files and its colours.

```
var converter = new fileReader(OBJFILE, function(){
    //do rest here
});
```

Methods:

- `getArray(): string[][][]`
    - Used to get the colour coordinates array.

RuleApplyer, making things look nice:

This class is used to convert the coordinate array from the first converter and creates a three-js group mesh from it using the specified rules files that are written in JSON format.

Rule file:

```
var ruleFile = JSON.parse(jsonString);
```

The rule file is a JSON file or string that specify how the colours in the model array should look as a 3D model. Here is an example of a complete rule:

```
{"Rules" :
    [{
        "Color" : "----" ,
        "Shape" : "----",
        "Texture" : "-----",
        "Bmap" : "-----"
    }]
}
```

The only rule needed is Color, the rest can be omitted and will then cause the default shape/textures to be apllied.

Using the class:

```
var ruleApplyer = new RuleApplyer();
ruleApplyer.convert(ruleFile, model);
converter_model = ruleApplyer.output();
demo_canvas.setMesh(converter_model);
```

Attribute:

- `private theArray : string[][][];`
- `private theMesh : THREE.Group;`
- `public interpreter : RuleInterpreter;`

Methods:

- `constructor()`
- `convert(ruleFile : RulesFile, inputArray : string[][][]) : THREE.Group`
- `output() : THREE.Group`