

# Threejs OpenJSCAD Robot Arm – Sam Ross

## Task Outline

Programmatic design – OpenSCAD (and equivalent for electronics and possibly control software) different design choices for the parts of the robot are expressed as parameters and cad models and related parts can be automatically constructed for each parameter set (to enable a search for an optimal design for different situations)

Ultimate goal – Make a parameterizable function that outputs a robot arm that we can manufacture and simulate.

## Current Stage

Link to my Github Pages which has the OpenJSCAD and Threejs conversion:

<https://sam-ross.github.io/OpenJSCAD-Threejs/templates/index.html>

The link above should take you to the demo page of the subproject. The page consists of two iFrames: one for the OpenJSCAD page (on the left) and one for the Threejs page (on the right).

**OpenJSCAD** is a set of modular, browser and command line tools for creating parametric 2D & 3D designs with JavaScript code.

**Threejs** is a cross-browser JavaScript library and application programming interface used to create and display animated 3D computer graphics in a web browser using WebGL.

**Ammo.js** is a direct port of the Bullet physics engine to JavaScript, using Emscripten. The source code is translated directly to JavaScript, without human rewriting, so functionality should be identical to the original Bullet.

As seen in the demo, OpenJSCAD robot arm on the left is created entirely from the OpenJSCAD code found in the tab to the right of the page. To export the OpenJSCAD robot into the Threejs page click on the “Generate STL” button when “Threejs Export – STL (ASCII)” option is selected.

Each part of the robot arm should load into the Threejs scene one by one and once the final mesh is loaded in, multiple balls will drop from above, to collide with the arm, hence showing that the collision shapes are all working as intended.

# Getting everything set up

The following guide is part of a guide called Mech\_Turk\_Dummys\_Guide\_-\_hugh\_13-04-20 and was written by Hugh .

---

## Installing Tools

1. Install Python:  
(not sure if this is needed because anaconda will come with python???)  
<https://www.python.org/downloads/>
2. Install VS Code:  
(can also be done after step 3, inside the Anaconda Navigator)  
(download from here. I recommend the system installer over single user)  
<https://code.visualstudio.com/Download>  
  
and install the Python language support extension:  
(click link on right side of welcome screen, search in extension store or use this link)  
<https://marketplace.visualstudio.com/items?itemName=ms-python.python>
3. Install Anaconda:  
(download the individual edition here)  
<https://www.anaconda.com/distribution/>  
(in the advanced option screen, check 'Add Anaconda to my PATH environment variable')
4. Set up a separate python environment and link your vs code to it:  
(this article guides you through:  
setting up a new environment using the anaconda navigator,  
adding the new environment to the path and  
then selecting that environment to use as your python interpreter in your VS)  
<https://medium.com/@akhilsai831/setting-up-anaconda-environment-with-visual-studio-code-in-windows-10-ac3f9afd80e0>  
(note, you will need to have a python file or folder open in VS code in order to change the interpreter)  
[\*\*\* If you launch VS code from the anaconda navigator, you can skip some steps in the article \*\*\*]  
<https://docs.anaconda.com/anaconda/user-guide/tasks/integration/vscode/>  
"When you launch VS Code from Navigator, VS Code is configured to use the Python interpreter in the currently selected environment."
5. Install Flask  
(careful following the official Flask guide here)  
<https://flask.palletsprojects.com/en/1.1.x/installation/>

### INSTEAD:

- a. Open command prompt on your computer.
- b. Navigate to your python workspace directory (whichever folder you're working in) (to see what is in your current directory type **dir** and hit enter, to change directory type **cd** followed but the file-path of your chosen directory).
- c. To create the venv folder (for virtual environments) type **py -m venv venv** and hit enter
- d. To activate it type **venv\Scripts\activate** and hit enter.
- e. To install flask with anaconda instead of pip type **conda install -c anaconda flask** and hit enter.  
(if it fails to install on the command line, use the anaconda navigator. Go to environments and click on your current one. Above the list of modules, select 'all' from the dropdown and then search for 'flask'. Check the box next to it and hit apply.)

### Using Flask

<https://flask.palletsprojects.com/en/1.1.x/quickstart/>

---

For setting up the flask server, create a file called "app.py" in your py38 folder, which should be at a directory similar to mine, shown below:

C:\Users\<username>\.conda\envs\py38

Next, paste the following code into your app.py file:

```
import mimetypes

mimetypes.types_map['.js'] = 'application/javascript'
print(mimetypes.guess_type('style.js'))

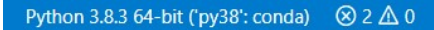
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/hello')
def hello_world():
    return 'Hello, World!'

if __name__ == "__main__":
    app.run(debug=True)
```

Once you have created this file called “app.py” you are ready to start up the server. Firstly, ensure that you have the correct interpreter path selected by looking at the bottom left of the screen. It should look similar the image shown below.

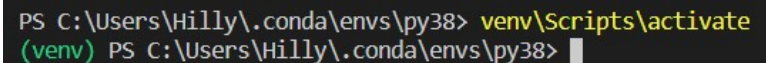


If the long blue bar doesn’t come up at the bottom of your screen, ensure that python is selected as the default language. You should also install any python extensions it asks you for.

To enable the virtual environment now, and at any future point click Terminal > New Terminal, then type:

```
venv\Scripts\activate
```

This should activate the virtual environment, as shown below.



Next, to run the app.py file, type:

```
python app.py
```

This should run through the code and start the server on the default address which should be the same as or similar to <http://127.0.0.1:5000/>

You should see an error now because the index.html file has not been added yet, but you can still check if the server is working. In the app.py file, the part of the code shown below means that when “hello” is added to the end of the default web address (<http://127.0.0.1:5000/hello>), a message should be displayed.

```
@app.route('/hello')
def hello_world():
    return 'Hello, World!'
```

A “Hello, World!” message should be displayed on the page, as shown below.



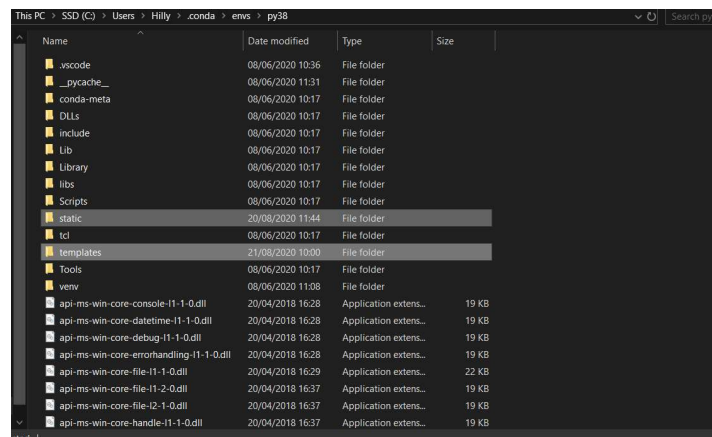
Now that the flask server is working, you need to add some files to the virtual environment folder.

Head over to my Github OpenJSCAD-Threejs repository at the following link and clone the master branch:

<https://github.com/sam-ross/sam-ross.github.io/tree/master/OpenJSCAD-Threejs>

Extract the zip and navigate to the “OpenJSCAD-Threejs” folder, where you will see two folders called “static” and “templates” and either copy or cut these folders.

Next navigate to your py38 folder (which you created app.py in) and paste the static and templates folder into this folder, as shown below.



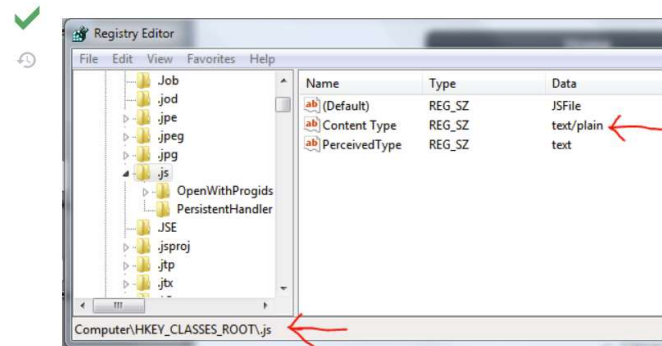
The **templates** folder is used to store only the main page that is loaded in the flask server.

The **static** folder is used to store basically every other file including: JavaScript files, CSS files and even html files that are used in iFrames created from the main page etc.

Currently there is a small issue with the flask server where the content-type of the js files is sent to the browser (chrome in my case) as **text/plain** instead of **text/javascript**. Without the workaround, javascript files that use ES6 modules won't load and will give a mime type error. To fix this error, follow the guide in the screenshot shown below ([source](#)):

195 The Visual Studio installer must have added an errant line to the registry.

open up `regedit` and take a look at this registry key:



See that key? The Content Type key? **change its value from text/plain to text/javascript**.

Finally chrome can breathe easy again.

Now that the static and templates folders are in place and the mime type workaround is in set up, if you head over to the flask server default base link (not including “hello” at the end ie. <http://127.0.0.1:5000/>) you should see that the index.html page has loaded, and it should look just like the demo from earlier.

## Understanding the code

Before I start explaining the code, there are a few files that you should have open. Below is a list of those files and their corresponding folder locations:

(The following file folder directories are relative to the py38 folder which is found at C:\Users\<username>\.conda\envs\py38)

- |                            |  |
|----------------------------|--|
| 1. app.py -                | py38   |
| 2. index.html -            | py38\templates                                     |
| 3. OpenJSCAD.html -        | py38\static\htmlPages                              |
| 4. Threejs.html -          | py38\static\htmlPages                              |
| 5. loadSTL.js -            | py38\static\js                                     |
| 6. STLloaderOpenJSCAD.js - | py38\static\three\examples\jsm\loaders             |
| 7. index.js -              | py38\static\OpenJSCAD.org-master\packages\web\dist |



Above is how it would look if you opened all these files in Visual Studio Code. Personally, I have the first 5 opened in VSC and have 6 and 7 open in Notepad++ as I found Notepad++ handier to me for navigating source code but this is just a personal preference.

### 1. app.py

You should already be familiar with app.py as you created it earlier in this guide. As a reminder, app.py is the python file that contains the code that runs the flask server in the virtual environment (venv). To start the venv and run the flask server in it, click Terminal > New Terminal, then type:

```
venv\Scripts\activate
```

This should activate the virtual environment, as shown below.

```
PS C:\Users\Hilly\conda\envs\py38> venv\Scripts\activate
(venv) PS C:\Users\Hilly\conda\envs\py38>
```

Next, to run the app.py file, type:

```
python app.py
```

It should then give you a link to the address that the server is run on (<http://127.0.0.1:5000/>) as shown below:

```
PS C:\Users\Hilly\.conda\envs\py38> venv\Scripts\activate
(venv) PS C:\Users\Hilly\.conda\envs\py38> python app.py
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 241-710-646
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

## 2. index.html

Index.html is the html page that is displayed when the flask server is running. In this html file, (between the script tags) is where the global variables are declared. These global variables are created so that the pages in the iFrames (3&4) have a way of communicating as they currently don't have any other way. These global variables can be accessed by adding the word "parent" in front of the identifier.

For example, to access the "counter" variable in loadSTL.js (5) you would type "parent.counter" as shown below on line 248:

```
248 | | | parent.counter++;
```

Before the sub project was at its current stage (in the demo), the OpenJSCAD side and the Threejs side of the Github Pages demo were separate html pages (3&4). For getting those two pages to display on one page, I decided to go with iFrames (although it may not be the most effective way of doing so).

An iFrame (Inline Frame) is an HTML document embedded inside another HTML document on a website. The iFrame HTML element is often used to insert content from another source, such as an advertisement, into a Web page.

In this case, the iFrames were used side by side and the source of what is displayed in them are the html pages: OpenJSCAD.html (3) and Threejs.html (4).



### 3. OpenJSCAD.html

**OpenJSCAD** is a set of modular, browser and command line tools for creating parametric 2D & 3D designs with JavaScript code.

OpenJSCAD.html is the html page that is displayed in the left iFrame in the demo. The original html file was downloaded from github and the html page itself is very similar to the main OpenJSCAD website at <https://openjscad.org/>.

For most of the backend code of the web page, we have index.js (7), which is called in the script tags of OpenJSCAD.html and runs alongside it.

There are a few useful shortcuts to take note of. These shortcuts only work **when the cursor is on the code tab on the right**:

- Ctrl + S                                      Saves to the local cache
- F5 or ( shift + return )                  Renders the code (ie. reloads the cad design)
- CTRL + SHIFT + \                          Clears the cache

You should be aware of the cache because if you, for example, upload the project to Github Pages and have a cache saved, then what code/model you see will be different from someone with no cache. You can double check how it looks for other people by loading the Github Pages page in private browsing/incognito mode.

By default, the OpenJSCAD page will try to load the code and render it from py38\static\OpenJSCAD.org-master\packages\examples. The code for this is on line 94835 and 95029 in the index.js file (7).

I have added the robot arm jscad file (robotArmUpdated) to this examples folder and set it as the default . You can create and download a jscad file by exporting as jscad type from the OpenJSCAD page on the flask server or by going to the official website at <https://openjscad.org/>.

This example file is only loaded if the cache is empty ie. the example file will be loaded for new users so be aware to differentiate between the cache and the default example file.

On line 130 (in the editor div), I believe is placeholder code for the page, meaning that this code will be there incase there is no cache or example to be loaded. I'm not entirely certain of this though.

Finally there is the export selection list. The exports should work fine in that they should allow you to download an export file of that type. The export type "Threejs Export - STL (ASCII)" used to be just called "STL (ASCII)" until the process was edited so that the model was loaded into the Threejs page. This will be gone into in depth in the index.js (7) section.

On the left tab I added a text area which can be seen with the placeholder text "Physics Data" and also added a form submit button. As of now, this feature does nothing as it has not been implemented but will be explained in the "What's Next" section later.

## 4. Threejs.html

**Threejs** is a cross-browser JavaScript library and application programming interface used to create and display animated 3D computer graphics in a web browser using WebGL.

**Ammo.js** is a direct port of the Bullet physics engine to JavaScript, using Emscripten. The source code is translated directly to JavaScript, without human rewriting, so functionality should be identical to the original Bullet.

Threejs.html is the html page that is displayed in the right iFrame in the demo. This page is used to display the Threejs scene, which the OpenJSCAD model is loaded in to.

As seen in the code for Threejs.html, there are two script tags. The first script tag is for loading the code for Ammo.js so that all the Ammo functions can be used. The second set of script tags is for loading in the loadSTL.js file (5).

This JavaScript file is loaded in as a JavaScript ES6 module (as seen by where it says type="module"). The reason for it being loaded as an ES6 module will be explained in the loadSTL.js (5) section below.

## 5. loadSTL.js

LoadSTL.js contains all the code for the Threejs scene, so this is the most currently relevant file as this is where most of the current work is being done. The scene, camera, lighting, etc. are all set up when the page is initially loaded. Once the "Generate STL" button is clicked, the OpenJSCAD side exports the OpenJSCAD robot arm as stl text in a string and the STLloaderOpenJSCAD.js (6) file converts this stl text into Threejs objects.

For the explanation of the loadSTL.js file, I will go through it in an order that should make the most sense.

First of all, there are the import statements at the top of the file for importing the files needed. The use of these import (and also export) statements is only allowed in JavaScript ES6 modules. In this case, Threejs is imported as a module.

The Orbit Controls and STLloader files are modules (as they use import and export statements), so to use them, they must be imported as modules (by using import statements to import them). To allow for this (ie. to use loadSTL.js as an ES6 module) loadSTL.js is set to **type="module"** in Threejs.html. Threejs can be used as a non-module or module but in this case it is used as a module because the loadSTL.js file needs to be set to type module anyway – so that orbit controls etc. can be used.

Below all the import statements in LoadSTL.js, there are the local variable declarations for the file. Line 18 is where ammo is initialised and here the **start** function is called. The start function runs through a list of methods that basically sets up the Threejs scene.

One of the methods called in the start function is `setUpEventHandlers` and it is used to set up the detection of the keys for moving the parts of the robot arm. This feature is not fully finished being set up until the full robot arm has been loaded later.

The final function to be called in the start function is **animate**. Animate is the function which is constantly reiterating over and over again. Each time it runs through, it does all the physics calculations to work out how all the components of the scene (ie. the objects, camera, lighting, etc.) should be changed for the next frame, and then this new frame is displayed. These frames are constantly being calculated then displayed for the user.

On line 681 there is an if statement in `animate` which basically checks if the stl text from `index.js` has been exported. Animate is constantly being looped through, so this if statement is being trialed constantly. Once the export has gone through successfully, `parent.export_stl` (a global variable) is no longer null (as it contains all the stl text) so the if statement runs through successfully. The variable `hasBeenRunOnce` is used to basically make sure that the if statement is only run once for each part that is imported. Then finally the **loadNewSTL** function is called.

`LoadNewSTL` can be found around line 104 and firstly it uses the custom `STLLoaderOpenJSCAD.js` (6) file to take in the stl text and creates **Threejs meshes** for each of the parts out of it. These Threejs meshes are only for visuals and they currently have no ammojs physics properties so they will not collide with other meshes and objects. These meshes aren't added to the scene until line 172.

From around line 130 to 160 is the code for setting up the automatic adjustment of the orbit controls target so that once the full robot arm is loaded in, the orbit controls are centered about the center of the robot arm in every axis.

Line 178 is where the collision shape is set to the return of the call of the `getObjectCollisionShape` method. This method can be found around line 301 and it basically works by taking the x,y,z coordinates of the 3 points of every triangle that makes up the outside mesh and creates an **ammojs** collision shape out of it. The collision shape is basically the shape that is "seen" by the collision/physics engine.

Once the collision shape is returned, it is used with all the ammojs code from line 186 to eventually create and add the rigid body for the robot arm part on line 225.

The next 5 if statements, starting at line 231 are used to create a constraint between the mesh just created and the last mesh created. The first constraint is a fixed constraint between the base and the bottom cylinder. The second constraint is a hinge constraint about the y axis so the robot arm can be rotated in a circle. The rest of the constraints are hinge constraints about the x axis.

The rest of the code of the `loadNewSTL` function (ie. lines 278-291) is basically just code to reset all the variables and such so that another part can be loaded in. It also checks if this is the last part to be loaded in and if it is then the balls will be dropped and the rest of the code for the movement of the parts will be set up (line 661).

The movement of the parts of the arms with the hinges doesn't quite work as intended and will this will be explained in the "What's Next" section later.

## 6. STLloaderOpenJSCAD.js

**STLloaderOpenJSCAD.js** is the modified version of **STLloader.js** which can be found in the same location. The original **STLloader.js** file is a Threejs loader which allows the user to load in Threejs meshes from an stl text file. The reason behind creating the modified version (**STLloaderOpenJSCAD.js**) was that **STLloader.js** could only load in stl text from an stl file, whereas I needed it to take direct text input somehow from index.js (7).

The only real change made can be seen at the start of the code in the load function where the actual stl text is passed as a parameter instead of a url to an stl file.

The global variable, export\_stl (or parent.export\_stl when using it outside of index.html), is used to firstly store the stl output from index.js (7). It is then used in STLloader.js (5) - as a parameter for the STLloaderOpenJSCAD.js call on line 111.

## 7. Index.js

Index.js is the file that is responsible for most of the backend work for the OpenJSCAD.html page. The section that I focused on changing was the export system for ASCII STL.

On line 93440 is the function for when the export file is clicked. Before I changed it, all that used to be in that function was “that.generateOutputFile();”. It has been changed so that for the number of parts (ie. Length of the OpenJSCAD code “objects” array on the actual page) it will run through the export process. The for loops uses the setTimeout function to add a delay of 400ms between each call of that.generateOutputFile. Without the timeout, there wasn’t enough time for each part to be exported into the Threejs scene.

This button click function needed to be changed as index.js used to export the whole robot arm as one stl file whereas I needed to export each part individually – so that each part was loaded into Threejs separately. There is more code changed to allow for this, which I will go through in this section.

Line 225 is where the parent.openjscad\_code variable global is given its value.

Line 403 is where the drop-down menu selection was changed from **STL (ASCII)** to its current text which is **Threejs Export - STL (ASCII)**.

Line 490 is where the parent.object\_array\_length global variable is assigned its value.

Line 536 is the replacement for the currently commented-out line on line 535. Line 535 basically called a method that took all the objects in the OpenJSCAD code object array and merged them all together into one stl file. In the new line 536, the variable object is just set to the current object in the array, instead of merging each part.

Line 19270 is the function where the stl file is created. Line 19281 is the line where the parent.export\_stl global string variable is assigned the string value of the completed stl file for this robot arm part. Now that this variable has a value, the if statement in the animate function in

loadSTL (5) will now be true and will run, and hence, the Threejs meshes will be created using this string as a parameter.

This whole process will be repeated for the number of objects in the object array as defined on line 1 of the OpenJSCAD compiler code (tab on the right side of OpenJSCAD.html iFrame page).

Index.js is also where the example that is shown on the OpenJSCAD.html page is specified. I exported the robot arm I created in the OpenJSCAD code panel as a **JSCAD** file and copied it to the examples folder – which can be found at “static\OpenJSCAD.org-master\packages\examples”.

Then to set this file as the default example, I went to line 94835 and changed the first file in the examples array to my file name which is “robotArmUpdated.jscad”.

Line 95029 is where this example file is set as the default file to be loaded in.

## What's Next?

The high level goal for this sub-project is to create a parameterizable function that outputs a robot arm that we can manufacture and simulate. Then we'll do a similar thing for OpenSCAD (different from OpenJSCAD) for calibration objects and for testing of the arms – as in how strong it is and any other tests needed. Will use the parameterised features of OpenSCAD to create all these slightly flawed versions, which are plausible that we would actually make so we can run the experiment with them all. So then theoretically, we'll be able to do a search over the designs of the robot to get the one that solves our tests as best as possible. Then eventually at some point manufacture the thing.

If you've been playing around with the robot arm in Threejs you'll probably have realized that hinges 3-5 work fine until you move hinge 2. What appears to happen is that the axis of rotation for hinges 3-5 don't seem to update as the robot arm is rotated around hinge 2, so hinges 3-5 rotate about the same axis and direction, and hence, they rotate at crazy angles.

The problem shown above is just the surface level of the problems of the constraints. When the parts are joined together by the hinges they should move about like a fluid ragdoll but instead they stay stuck stationary in the air which is not how they should be. I don't think the issue is because of spacing/gaps between the parts but I think this should be easily solved as the constraints can be broken down into very simple examples ie. two cuboids joined together by a hinge constraint.

In all of the movePart methods in loadSTL.js (starting at line 620), an impulse should be applied to the parts, instead of the angular velocity being set to a certain value (using setAngularVelocity).

Where I was planning on working to next :

1. Get ragdoll parts working (I may have set up the hinges wrong) (can simplify down to cuboids etc.)
2. Try to get an understanding of how the hinges work and get a printable output by placing breakpoints where the hinge constraints are added in. There should also be a call back method for constraints to output details about the constraint
3. Try to work through figuring out how the rotation matrix and other code works and how it updates and work towards getting all the hinges to automatically update their current axis as hinge 2 spins round

Once the robot arm is set up with constraints that are working as expected, the loadSTL.js code should be edited so that the methods for creation of any constraints can be reused and will have parameters which take values passed in from the OpenJSCAD.html side.

There is a textarea in the left panel that is not currently functional, but shows an example of where this data can be passed from. This data should be used to create a JavaScript dictionary of all the constraints that need applied. So have it so that the text in the textbox will be read as a JSON which is read as a string and then turned into a JS dictionary in the html. This will store the metadata information about the objects ie. their physics properties and what the constraints are on the physics side.