# render-python Documentation

***Release 1.0.1***

**Forrest Collman, Russel Torres, Eric Perlman**

**Jul 19, 2023**

# Contents:

CHAPTER 1

# API

## 1.1 User Guide

### 1.1.1 Getting Started

First you must have a render server running and accessible to you. For instructions on getting a render server up and running http://www.github.com/saalfeldlab/render

For the purposes of this tutorial i'm going to assume you have a render-server installed and up and running on localhost and your render dashboard is viewable at http://localhost:8080/render-ws/view/index.html

You also should have available on your local system a path to the render-ws-java-client scripts with a compiled jar file. One way to get these things, with render-python already installed is to use get the render-python Docker image (fcollmanrender-python), where it will be installed at usrlocalrenderrender-ws-java-clientsrcscripts

```
$ docker pull fcollman\render-python
$ docker run -t fcollman\render-python \
    -v .:\scripts \
    python \scripts\my_script.py
```

### 1.1.2 Making a new stack

First we have to setup our default connection properties for our render server

```
import renderapi

#create a renderapi.connect.Render object
render_connect_params ={
    'host':'localhost',
    'port':8080,
    'owner':'myowner',
    'project':'myproject',
```

(continues on next page)

(continued from previous page)

```
    'client_scripts':'\usr\local\render\render-ws-java-client\src\scripts'
    'memGB':'2G'
}
render = renderapi.connect(**render_connect_params)
```

You can simplify your call to `renderapi.connect()` if you wish by setting up some or all of the following environment variables DEFAULT_RENDER_HOST, DEFAULT_RENDER_PORT, DEFAULT_RENDER_CLIENT_SCRIPTS, DEFAULT_RENDER_PROJECT, DEFAULT_RENDER_OWNER, RENDER_CLIENT_HEAP.

Now we can create a new stack on the render server

```
#make a new stack
stack = 'mystack'
renderapi.stack.create_stack(stack,render=render)
```

now you should be able to see your stack at http://localhost:8080/render-ws/view/stacks.html. It is presently in the LOADING state as it is awaiting tiles to be loaded. In order to give it some tiles you must fill out some metadata information about the tile. Many fields are technically optional, but it's best to fill them out if you can. In this example we will do it manually, of course most users will likely write code to create this metadata from existing metadata.

```
#define a tile layout
layout = Layout(sectionId='1',
                scopeId='myscope',
                cameraId='mycamera',
                imageRow=0,
                imageCol=0,
                stageX=100.0,
                stageY=300.0,
                rotation=0.0,
                pixelsize=3.0)
```

Next you have to define the set of transformations that should be applied to the raw image. In this example we will use `renderapi.transform.AffineModel`. However, you can use any of the transforms defined in `renderapi.transform` or any transformation in the mpicpg library (https://github.com/axtimwalde/mpicbg) installed with your render server via the `renderapi.transform.Transform` class, provided you know its classname and dataString.

```
#define a simple transformation, here a translation based upon layout
at = AffineTransform(B0=layout.stageX/layout.pixelsize,
                     B1=layout.stageY/layout.pixelsize)
```

Now we can define the actual tile

```
tilespec = TileSpec(tileId='000000000000',
                z=0.0,
                width=2048,
                height=2048,
                imageUrl='/data/images/0_0_0.tif',
                maskUrl=None,
                layout=layout,
                tforms=[at])
```

Note that the path to the imageUrl needs to be a path that is readable by the render server if you want server side rendering of images to function correctly. If you only want to use render to store metadata information then this isn't strictly necessary, but other web-services such as https://github.com/neurodata/ndviz will require this.

### 1.1.3 Importing tilespecs

Now we can import the tile to our stack using `renderapi.client` which uses the render-ws-java-client scripts to perform client side validation and bounding box estimation of your tiles before uploading them to the server.

```
#use the simple non-parallelized upload option
renderapi.client.import_tilespecs(stack,
                                  [tilespec],
                                  render = render)

#now close the stack
renderapi.stack.set_stack_state(stack, 'COMPLETE', render = render)
```

If you have many tilespecs to import, it often makes sense to parallelize the client side validation and bounding box estimation. So lets simulate the importing of many tiles

```
rows = 10
cols = 20
sections = 500
overlap = .8 #20% overlap
pix = 3.0 #nm
img_width = 2048 #pixels
img_height = img_width

tilespecs = []
for section in range(sections):
    for r in range(rows):
        for c in range(cols):
                layout = Layout(sectionId='%05d'%section,
                                scopeId='myscope',
                                cameraId='mycamera',
                                imageRow=0,
                                imageCol=0,
                                stageX=c*img_width*overlap*pix,
                                stageY=r*img_height*overlap*pix,
                                rotation=0.0,
                                pixelsize=pix)



                #define a simple transformation, here a translation based upon layout
                at = AffineTransform(B0=c*img_width*overlap,
                                     B1=r*img_height*overlap)

                tileId = '%d_%d_%d'%(section,r,c)

                ts = TileSpec(tileId=,
                              z=section,
                              width=img_width,
                              height=img_height,
                              imageUrl='/data/images/%s.tif'%tileId,
                              maskUrl=None,
                              layout=layout,
                              tforms=[at])
                tilespecs.append(ts)
```

This would of course would need to be adapted to suit the needs of your specific situation, but assuming you have a large number of tilespecs, they can be imported more efficently using `renderapi.client.`

`import_tilespecs_parallel()` which will also close the stack for you if you'd like.

```
renderapi.client.import_tilespecs_parallel(stack,
                                           tilespecs,
                                           close_stack=True,
                                           render = render)
```

When you are done, you should be able to see your stack on the render dashboard.

### 1.1.4 Transformations

The idea behind render is that it serves as a central place to store the metadata data about image tiles and how they should be tranformed. Central to that concept is what transformations it supports. Render is written in java and uses the mpicbg library (https://github.com/axtimwalde/mpicbg), the same library that backs TrakEM2, to perform all server side image transformation.

Render-python is client side library and can assist you in managing and setting those tranformations, and performing some calculations using the `renderapi.transorm` module. We have focused our initial efforts at supporting the most commonly used types of transformations.

Some transformation types presently support *tform* and 'inverse_tform' methods for calculating where numpy array sets of points map to and from these tranformations. Some presently support *estimate* methods which given a set of source and destination points, allow the estimation of a best fit transformation.

`renderapi.transform.estimate_dstpts()` simplifies mapping points through an ordered list of transformations. `renderapi.transform.estimate_transformsum()` provides a general way to produce a single `renderapi.transform.Polynomial2DTransform` that approximates a list of tranforms which have implemented *.tform* methods.

One aspect to keep in mind about render is that it supports `renderapi.transform.ReferenceTransform` which allows many tiles to share a common transformation without having to store its parameters directly. This is a conveient way to save on database storage and ensure many tiles are identically manipulated. However, depending on what you want to do, it can make things slightly more complicated.

For example, presently, the java client side scripts used to calculate bounding boxes and validate transformations need to have access to the referenced transformations in order to do their work, even if they already exist in the database. This is what the *sharedTransforms* argument in import methods within `renderapi.client` is for.

The `renderapi.transform.ReferenceTransform.tform()` method does not exist because the `renderapi.transform.ReferenceTransform` transform doesn't have any data about what kind the transform parameters are. Most render-python calls default to returning dereferenced transforms, which avoids this issue. However, this will break the efficency gains if you simply upload those dereferenced transforms. This is why `renderapi.tilespec.get_tile_spec_raw()` exists, in order to give you referenced transforms if you so wish. Also, you can use the `renderapi.client.importTransformChangesClient` to accelerate and simplify many transform modification tasks, as it won't do any client side validation or bounding box calculations.

### 1.1.5 Pointmatch database

Pointmatchs are locations between two images that correspond. Render has two groups of web services that are both available on the same web interface, the 'tile' based services and the 'pointmatch' services.

They are loosely coupled in the sense that they are stored in distinct databases, and make no assumptions about how the other is structured. This allows them to be deployed and maintained seperately, but can make things confusing if you assume that one knows more about what the other is doing than it does. To make things less confusing there is a set of reccomendations that you can read at *Pointmatch Assumptions*.

Pointmatches are stored by collection, group's and id's (see *reccomendation*) and have a source 'p' and a destination 'q', thus each set of matches in a collection is specified by a pGroupId, pId, qGroupId, qId combination. Functions in the `renderapi.pointmatch` module allow you to make queries on these point matches in various ways, and upload new matches. You might place some point matches between tile 0_0_0 on section 0, and tile 1_0_0 on section 1, using `renderapi.pointmatch.import_matches()` and then retrieve them using `renderapi.pointmatch.get_matches_from_tile_to_tile()`.

```
matches_in={
     'pGroupId':'0',
     'qGroupId':'1',
     'pId':'0_0_0',
     'qId':'1_0_0',
     'matches':{
         'p':[[0,0],[1000,1000],[1000,0],[0,1000]],
         'q':[[0,0],[1000,1000],[1000,0],[0,1000]],
         'w':[1,1,1,1]
     }
 }
 renderapi.pointmatch.import_matches('mycollection',
                                    [matches_in],
                                    render=render)
 matches_out=renderapi.pointmatch.get_matches_from_tile_to_tile('mycollection'
                                                               '0',
                                                               '0_0_0,
                                                               '1',
                                                               '1_0_0',
                                                               render = render)

 print(matches_out)
 >> [{
     'pGroupId':'0',
     'qGroupId':'1',
     'pId':'0_0_0',
     'qId':'1_0_0',
     'matches':{
         'p':[[0,0],[1000,1000],[1000,0],[0,1000]],
         'q':[[0,0],[1000,1000],[1000,0],[0,1000]],
         'w':[1,1,1,1]
     }
 }]
```

### 1.1.6 Installation

from source

```
$ git clone https://www.github.com/fcollman/render-python
$ cd render-python
$ python setup.py install
```

## 1.2 Pointmatch Assumptions

Generally, I would reccomend adopting a set of conventions that constrain the relationship between data in the 'tile' and 'pointmatch' databases. These conventions are informed in large part based upon how the EMAligner https://github.com/khaledkhairy/EM_aligner assumes they are related, when using them to solve alignment problems.

### 1.2.1 tile_owners = pointmatch_owners

Particularly on deployments which have only a single render service, this will save you from having to redefine the default owner in the `renderapi.render.Render` or override it at each step. Make all stacks related to a dataset owned by a single owner, and make all pointmatch collections owned by that same owner.

### 1.2.2 groupId=sectionId and id=tileId

Or more verbosely, groupId/qGroupId/pGroupId = sectionId and pId/qId/id = tileId. Groups thus correspond to what section the point match is from and the id's correspond to what tile they are from. Technically, there is no need to make this association, and none of the render web services strictly require it. However, if you want to use the pointmatch results in combination with the tile services, it will be far easier if there is a strict mapping between these two databases. In addition, tools like EMAligner and ndviz are presently written in a way that assumes this mapping is held, so you need to make the same assumption if you want to use those services.

### 1.2.3 Know your 'local'

Write all pointmatches between tiles in a consistent 'local' coordinate system and make that local coordinate system the raw image space given by rendering the tile using `renderapi.image.get_tile_image_data()` with normalizeForMatching=True and scale=1.0.

This would be conceptually simple, if the 'local' meant the same as `renderapi.coordinate` module defines local to mean, namely the raw image space, with the upper left hand pixel at 0,0 and positive x to the right and positive y down.

However, in some deployments of render this is not the case, and you might find that rendering a tile using normalizeForMatching=True does not produce a raw image tile. In fact it might render blank data in some circumstances if you have more than 1 transformation.

This is because, at Janelia the EMAligner was developed on TEM images that need a lens correction transformation, and the pointmatches are defined on the 'local' coordinate system after lens correction. This simplifies solving for the non-lens component of the transformation, as the EMAligner only needs to specify the single transformation that brings 'local' pointmatches into 'global' alignment and can safely disregard the non-linear effects of the lens correction.

However, it produces the confusing result that mapping the 'local' point matches coordinates through `renderapi.coordinate.local_to_world_coordinates()` does not give the correct result, and you have to have a second stack with lens correction transformations removed in order to map point match coordinates from local to world coordinates accurately using the coordinate mapping service.

More discussion on this at https://github.com/saalfeldlab/render/issues/13, https://github.com/saalfeldlab/render/issues/31.

I implemented an alternative strategy at https://github.com/saalfeldlab/render/pull/29 which adds a removeAllOption=True to many render calls which simply removes all transformations from the tilespec before returning or rendering. In applications where non-linear lens corrections are minimal, this simplifies things.

However for TEM or other applications with stereotyped non-linear transformations, it will make using the EMAligner to solve alignment problems more difficult, as the EMAligner doesn't know that it should map the pointmatches into the post non-linear correction space before attemptign to solve and isn't presently written to do this.

## 1.3 renderapi package

### 1.3.1 Submodules

### 1.3.2 renderapi.client module

### 1.3.3 renderapi.coordinate module

### 1.3.4 renderapi.errors module

### 1.3.5 renderapi.image module

### 1.3.6 renderapi.pointmatch module

### 1.3.7 renderapi.render module

### 1.3.8 renderapi.stack module

### 1.3.9 renderapi.tilespec module

### 1.3.10 renderapi.resolvedtiles module

### 1.3.11 renderapi.transform module

### 1.3.12 renderapi.utils module

### 1.3.13 Module contents

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search