Report

Image Dataset for atriculated cubiods

Richard Häcker

December 2019

Contents

1 Project

This software project was made in a project at the research Group Computer Vision at the Heidelberg Collaboratory for Image Processing. Special thanks to my tutor Johannes Haux and Konstantin Neureither who provided his project which served as starting point for this work.

1.1 Description

This Software Project enables the Linux and Windows user to create a custom dataset of labeled images with articulated cuboids. Either on your local machine or on an ssh server with "X11Forwarding". This is achieved through a two way communication of a TCP socket connection between the python client and the game engine Unity 3D.

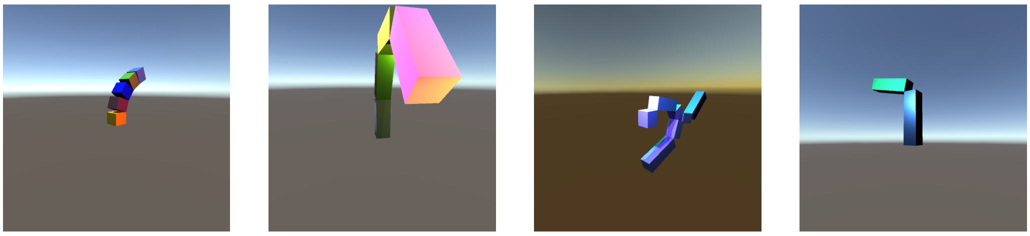


Figure 1: Some random generated Images

The data set can be created with the class `dataset_cuboids()` which initialises a client from the class `client_communicator_to_unity()`. The client then launches an unity executable and connects to it. With that a *C#* script starts in Unity and creates a TCP socket server to listen at a given port and host address for the client. Unity can then create 3D Objects and Scenes according to the received parameters which are rendered by a camera and send back to python.

1.2 Getting Started

If you just wish to get started right away without diving deeper into this project you do not have to look at the section Developers. I highly recommend to read this report up to the section 1.7.1 Simple Example. Additionally you can have a look into the Documentation if anything in the code is unclear.
If you want to get an in depth understanding of this project, keep reading and try out the coming Examples for yourself. Furthermore have a look at the section Developers.

1.3 First Steps

1. Download or clone my repository.
   a) Download from here.

   b) Or go to the directory you want to clone this project to and copy the following code into you terminal.

      ```
      git clone https://github.com/R-Haecker/python_unity_images.git
      ```
2. Try out the examples given in section Examples and modify them as you desire.
   - You can have a look into the Documentation of the code and the meaning of all functions as well as their parameters here.

1.4 Prerequisites

- You need all of the following packages installed:
- Both Files:
  – PIL, numpy, json, logging, os, time
- dataset.py:
  – matplotlib, tkinter, math, copy,
- client.py:
  – socket, sys, io, subprocess, inspect,

1.5 Coordinate System

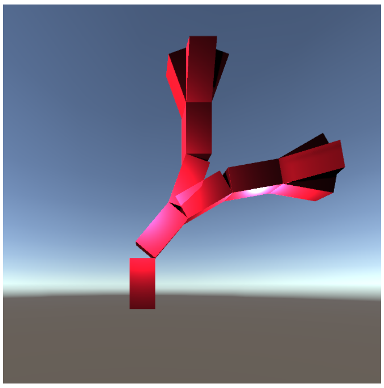
1.5.1 Cartesian Coordinate System

In the picture we are looking at the **y-z plane**. The **x axis** is facing towards the camera, the **y-axis** is positive in the vertical direction and the **z axis** going left to right horizontally. This is the definition of the coordinate system of Unity.

1.5.2 Spherical Coordinate System:

The parameters used in this project are all in spherical coordinates and the angles are specified in degrees. The coordinated system follows the standard convention as described here. On this link the only difference is that the y axis is named z in Uinty and the z axis is named y.

1.6 Parameters




```json
{
    "total_cuboids": 5,
    "same_scale": true,
    "scale": 2,
    "same_theta": true,
    "theta": 20,
    "phi": 90,
    "total_branches": [1,2,1,4],
    "same_material": true,
    "metallic": 1,
    "smoothness": 0.5,
    "r": 1,
    "g": 0.1,
    "b": 0.2,
    "a": 1,
    "CameraRes_width": 1024,
    "CameraRes_height": 1024,
    "Camera_FieldofView": 100,
    "CameraRadius": 10.0,
    "CameraTheta": 90,
    "CameraPhi": 0,
    "CameraVerticalOffset": 0,
    "Camera_solid_background": false,
    "totalPointLights": 2,
    "same_PointLightsColor": true,
    "PointLightsColor_r": 1,
    "PointLightsColor_g": 1,
    "PointLightsColor_b": 1,
    "PointLightsColor_a": 1,
    "PointLightsRadius": [5,6],
    "PointLightsTheta": [45,60],
    "PointLightsPhi": [0,95],
    "PointLightsIntensity": [10,12],
    "PointLightsRange": [10,10],
    "totalSpotLights": 0,
    "SpotLightsRadius": null,
    "SpotLightsPhi": null,
    "SpotLightsTheta": null,
    "SpotLightsIntensity": null,
    "SpotLightsRange": null,
    "SpotLightsColor_r": null,
    "SpotLightsColor_g": null,
    "SpotLightsColor_b": null,
    "SpotLightsColor_a": null,
    "same_SpotLightsColor": null,
    "SpotAngle": null,
    "DirectionalLightTheta": 60,
    "DirectionalLightIntensity": 3
}
```


Figure 2: Crane and Parameters

On the right hand side you can see a picture of **articulated cuboids**. The whole structure is named a crane. Beneath it you can see the parameter as Json file which defines all properties of the scene. Both of them can be saved to reuse later or to be a part of the data set you want to use. Here are only the important parameters mentioned. If you want to use all parameters look at my documentation.

Parameters:

- **total_cuboids=5**, This means that along one "branch" there are five cuboids stack on top of each other.
- **scale=2, same_scale=True,** This means that all cuboids will have the scale two in the original vertical direction.
- **theta=20,same_theta=True,** Theta is the angle between two stacked cuboids. Here all the thetas are set to twenty degrees. If `Phi=0` the pivot axis or "hinge" of the first cuboid would be in the z axis.
- **Phi=90,** The rotation of the whole crane around the vertical axis at the origin. If we look at the Phi value of the camera: `CameraPhi=0` it makes sense that we get the side view of the crane. If phi is zero the cuboids pivot towards the x direction i.e. the camera.

- **total_branches=[1,2,1,4],**

  This parameter specifies the amount of branches created at every cuboid. This means if we want a simple crane with one branch it should be: `[1,1,1,1]`. This will be done in the following examples. Here we create two branches at the second cuboid and four branches with the last cuboid.

- **Material properties:**

  These parameters all define the material properties of the cuboids:
  r (Red), g (Green), b (blue), a(alpha), metallic, smoothness.
  If `same_material=True` all cuboids will have the same material otherwise every parameter has to be specified for every cuboid. The first four parameters mentioned specify the color of the material between zero and one. The fourth is the alpha channel or the transparency of the color. If you want to learn more have a look at the Unity documentation.

- **Camera:**

  The camera position is specified with these parameters in spherical coordinates:
  CameraRadius, CameraTheta, CameraPhi, CameraVerticalOffset.
  With the option to offset the origin of only the camera coordinates. Furthermore there are parameters for the metadata of the camera.

  CameraRes_width, CameraRes_height, Camera_FieldofView, Camera_solid_background.
  With the first two being the resolution of the receiving image, the third is the field of view of the camera and the last parameter determines if the camera should render a solid color in the background of the crane or the default skybox of Unity.

- **DirectionalLight**

  This is a light which is intended by Unity to represent the sun.
  DirectionalLightTheta=60, DirectionalLightIntensity=3
  Theta is the angle from where the sun light is coming. If Theta is ninety degrees the sun will set on the horizon in the negative direction of the z axis. This means that the directional light can only move in the z-y plane. The Intensity describes how bright the light is.

- **Spotlights and Pointlights**

  Every light has a position specified in spherical coordinates and a color. The color of all Spotlights or Pointlights can be set the same with for example `same_PointLightsColor`. Every light has the properties of intensity and range. Additionally the Spotlights have a property `SpotAngle` which defines the angle of the cone of the created light.

If you want a definition for all parameters have a look into my documentation.

1.7 Examples

The following section is also available in the read me file at my repository:

1.7.1 Simple Example

The following code represents a simple example how to use this project. This code creates and plots eight randomly generated images.

```
import dataset
data = dataset.dataset_cuboids(dataset_name = "simple_example")
dictionaries = []
for i in range(8):
    dictionaries.append(data.get_example(save_para = True,
                                         save_image = True))
data.exit()
data.plot_images(dictionaries, save_fig = True)
```

- In the first line the file dataset.py is imported.
- The first thing you want to do is to initialize an object of the class `dataset_cuboids()`.
  - This starts Unity and connects to it with code from `client.py`.
  - make sure that the string `dataset_name` does not contain any white spaces.
- After that you can use it as you desire.
- In this exampel we create random images with the function `get_example()`.
  - This function returns an dictionnary with the keys: `index`, `parameters` and `image`.
  - The argumets `save_para` and `save_image` are `True`. This means that the parameters of the created scene are saved inside a unique folder for your dataset. This folder can be found in `data/dataset/`, it is named with a time stamp and the name of your dataset specified in `dataset_name`. The parameters and and images are saved separately. You can have a look at the saved data of the example above.
  - This is done eight times and every returned dictionary is save in a list.
- If we are done with requesting images you should always close and exit the Unity application and the connection with `exit()`.
- At last we want to have a look at the created images with the function `plot_images()`.
  - `save_fig = True` means that the resulting figure is saved at `data/figures`.
- That is it we have successfully created and saved images with a ground truth.

1.7.2 Advanced Example

```
import dataset
data = dataset.dataset_cuboids(dataset_name = "advanced_example",
                               unique_data_folder = False)
data.reset_index()
dictionaries = []
for i in range(10):
    dictionaries.append(data.get_example(save_para=True,
                                         save_image=True))
data.exit()
data.plot_images(dictionaries, images_per_row=5, save_fig=True)

data2 = dataset.dataset_cuboids(dataset_name = "advanced_example",
                                unique_data_folder = False)

data2.set_config(total_cuboids=[2,3],same_theta=False,
                 DirectionalLightTheta=[80,90],
                 totalPointLights=None, totalSpotLights=None)

dictionaries2 = []
for i in range(10):
    dictionaries2.append(data2.get_example(save_para=True,
                                           save_image=True))
data2.exit()
data2.plot_images(dictionaries2, images_per_row=5, save_fig=True,
                  show_index=False)
```

- The first block of code is pretty similar to the simple example
- The initialization differs by one additional argument.
  - `unique_data_folder = True` means that if you later save images or parameters your personal dataset folder will not include a time stamp. This enables another instance as `data2` with the same `dataset_name` to use the same dataset folder to store more and different images and parameters.
  - The folder in `data/dataset/` will now just be named: `advanced_example`.
- Since the simple example was executed befor this example the index is currently at nine. It is stored externally in `data/python/index.txt`.
- With the function `reset_index()` we now set it back to zero. Keep in mind that if you choose to set `unique_data_folder = True` and reset the index you can overwrite your old data.

- In the function `plot_images()` we specified the shape of the figure.
  - Since we created 10 images and `images_per_row = 5`, we now plot 2 rows of each 5 images next to each other.
- Now we create another `dataset_cuboids` instance and save into the same dataset folder. This block of code can also be executed in a different python file.
- We will also use the function `get_example()`, but first we want to personalize our boundaries and settings for the randomly generated images to create different images.
- The intervals, which define in what range a parameter can be generated are saved in the config of the instance of the class `dataset_cubiods()`.
- Every instance has a default config initialized wich can be changed with the function `set_config()`. You should habe a look into the Documentation at the function `set_config` and to learn what the specific parameters mean, you should go to the function `write_json_crane()` in the `client_communicator_to_unity` class.
  - In our example we choose with the argument `total_cuboids = [2,3]` that only two or three cuboids are created on the main "branch".
  - with `same_theta = False` we specified that all angels between cuboids should be different.
  - with `totalPointLights=None` there will not be any Pointlights.
  - with `totalSpotLights=None` there will not be any Spotlights as well.
  - with `DirectionalLightTheta=[80,90]` we specify that the only light the "Sun" will have the polar angle theta between 80 and 90 degrees which means there will be dawn.
- This time when using the function `plot_images()` we set `show_index = False` to not show the index displayed on top of the images.
- We now created a dataset with two different configs in the same dataset folder.

1.8 File Descriptions

```
python_unity_images
├── build_linux
│   └── unity_server_rendering_images.x86_64
├── build_windows
│   └── unity_server_rendering_images.exe
├── data
│   ├── dataset
│   │   ├── 2019-12-08 18:42 example
│   │   └── example
│   │       ├── parameters
│   │       └── images
│   ├── figures
│   ├── python
│   │   ├── client_tcp_config.json
│   │   └── index.txt
│   └── unity
│       ├── server_tcp_config.json
│       └── started.txt
├── log
│   ├── python_client.log
│   ├── python_dataset_and_client.log
│   └── unity.log
├── server_unity_image_generation
│   └── Assets
│       ├── Scenes
│       │   └── empty_scene_for_image_generation.unity
│       ├── class_structures.cs
│       ├── create_crane.cs
│       └── TCP_server.cs
├── client.py
└── dataset.py
```

Figure 3: Directory Tree

Beforehand you could see a directory tree of my repository with the relevant files of my project and now starting from left to right and top to bottom there is a short description for every file.

- **`build_linux`** and **`build_windows`**

  This folder contains the Unity build for Windows or Linux.
  A build is a executable program from Unity which executes the earlier given *C#* Scripts and creates the necessary server which can receive data and send images back.

- **`data/figures`**

  This folder is used by the member function plot_images() of the class `dataset_cuboids` from `dataset.py` to save the created figures.

- **`data/dataset`**

  Inside this folder all the images and parameters are going to be saved inside your personal folder. Here are two examples already created.

  - **`example/images`** and **`example/parameters`**

    The folder *images* and *parameters* is used by the member function save() of the class `dataset_cuboids` from `dataset.py` to either save the image of current Crane as png file or to save the defining properties in form of a json file.

- **`data/python`**

  - **`client_tcp_config.json`**

    This Json file saves the ip address and the port of the connected clinet TCP socket. Furthermore the data will be loaded and saved in the member function connect_to_server() of the class `client_communicator_to_unity`.

  - **`index.txt`**

    This index text file only saves one integer as a global variable which can be used across many requests for datasets. The `dataset_cuboids` class identifies every requested Crane with this index and uses it to distinguish the saved data.

- **`data/unity`**
    - **`server_tcp_config.json`**

      This Json file saves the ip address and the port of TCP server socket listener. Unity loads the data and listens at this port and ip address.
    - **`started.txt`**

      This text file only saves a boolean as a zero or one. The client sets the value to zero before starting Unity and when Unity is fully started it sets it to one which can then be read by the client. The client can afterwards continue to connect to the server.
- **`log`**
    - **`python_client.log`**

      This log file logs everything from the class `client_communicator_to_unity` into this file. It is useful for debugging when the class `dataset_cuboids` is not used.
    - **`python_dataset_and_client.log`**

      This log file combines the logger of the class `dataset_cuboids` and the logger from the class `client_communicator_to_unity`.
    - **`unity.log`**

      This is the log file from the unity program.
- **`server_unity_image_generation/Assets`**

  The folder **`server_unity_image_generation`** is the project folder for the Unity Editor.
    - **`class_structures.cs`**

      This *C#* script contains the class structures for personalized objects in Unity. This enables parsing the received json data to objects inside of unity which then can be used to create the requested scene. For more information look at section `class_structures.cs`.
    - **`create_crane.cs`**

      This *C#* script is used to build up the whole requested scene. It gets the data from `TCP_server.cs` and creates the needed cuboids and stacks them in the requested way as well as the wanted lighting etc. For more information look at section `create_crane.cs`.
    - **`TCP_server.cs`**

      This *C#* script creates the TCP server listener and is used to recive and send data to the python client. For more information look at section `TCP_server.cs`.

- **`client.py`**

  This python file contains the class: `client_communicator_to_unity()` with member functions to connect and communicate with Unity. More details in the documentation.

- **`dataset.py`**

  This python file contains the class: `dataset_cuboids()` which can create either specified parameters or in a specified intervals randomly generated parameters. With the class from `client.py` these parameters are used to create many images which can be saved and used as an data set. More details in the documentation.

2 Developers

2.1 Computing Platform

This project can either be run on an ssh server or your local machine.
If you want to run it on an ssh server you should make sure the server and your machine have "X11-Forwarding" enabled or a similar tool working. Unity needs a graphical window to render images which means that you have to export the application window from the server to your local display. This also brings a long launching time with it which means do not cancel the script it can take up to maximum 10 seconds to initialise the data set.
If you have problems with the python client connecting with the unity application on the server you could alter the `client_tcp_config.json` and the `server_tcp_config.json` to a different ip-address which is not in use.
For both platforms you can follow the guide in section First Steps.

If you feel comfortable working with this project and the Examples you can go even further and write your own python functions inside the calss `dataset_cuboids`.

For Example:

- You could manipulated fixed parameters and combine them with others or manipulated a certain set of parameters suitable to your desires.
- You can also combine Trigonometric functions as *Sin(x)* to use as inputs for parameters to create interesting shaped cranes.

Let your creativity run free and if you feel limited by the tools and function implemented in this project the following section Unity will help you to implement new features into your scene.

2.2 Unity

If you want to manipulate other objects or properties inside the scene additionally to what I provided in this project you will need Unity 3D.
Unity is free to use can be installed after registration for Windows, Mac and Linux.

1. Create an Unity account.
2. Follow this guide to install the Unity hub and afterwards the Editor.
3. If you have never worked with Unity 3D before you should start with an easy tutorial to get used to it. This site: https://learn.unity.com/ from Unity is a good starting point to learn Unity but there are thousands of tutorials on the web to choose from.

   You should learn how to use the Editor environment and the basics of scripting inside Unity.
4. If you feel comfortable with Unity you can import my project.
   - Inside the Unity hub go to *Projects* and in the upper right corner you can import Unity projects with the button *Add*.
   - Now navigate into the directory where you cloned my repository to. Select the folder `server_unity_image_generation` and click *OK*.
   - You can now look and use my Unity project inside the Unity Editor.
5. The Scene used in my project is mainly empty. There are only the camera and the directional Light with the scripts `create_crame.cs` and `TCP_server.cs` attached.
6. Have a look at the File Descriptions of the scripts and the code itself.

2.3 Scripts

All these scripts are used inside my unity application and are written in c sharp.

2.3.1 `class_structures.cs`

This C sharp file serves the purpose to creates classes to structure the various data needed in the Unity application.

`TcpConfigParameters`:
An object from this class can load a json file with a given ip and port which then the server can listen to.

```
[System.Serializable]
public class TcpConfigParameters
{
    public string host;
    public int port;
}
```

`JsonCrane`:
An Object from the following class can store all information the server received from the client which is then used to build up the requested scene.

```
[System.Serializable]
public class JsonCrane
{
    public int total_cuboids;
    public int[] total_branches;
    public bool same_scale;
    public bool same_theta;
    public bool same_material;
    public float phi;
    public float del_phi;
    public JsonCamera camera;
    public DirectionalLight DirectionalLight;
    public int totalPointLights;
    public PointLights[] point_lights;
    public int totalSpotLights;
    public SpotLights[] spot_lights;
    public JsonCuboid[] cuboids;
}
```

All the classes mentioned in `JsonCrane` are defined in this script. If you are intested have a look at the file itself.

2.3.2 create_crane.cs

This script is used to process all the information received from the client and build up the requested scene in unity.
For this task the main function `create_scene()` is used.
It will receive the data from the script `TCP_server.cs` and create as many instances of cuboids and lights as needed. The final amount of cuboids have to be calculated with the function `getTotalAmountofCuboids()` depending on the amount of branches.
After that the lights are positioned and the specific settings are applied. Now the cubes will be created as well as their associated material. First the main branch will be created and the branches will follow afterwards. Here to set up the cuboids in the right position, vectors and the rotation matrices are used. These rotation matrices are defined in the functions: `rotVec_yz(), rotVec_xz(), rotVec_xy()`
which rotate a vector in the mentioned plane. At last the camera position and the specified parameters are set.
This function `Update()` is called once per frame and if the Boolean `ready_to_build` from the script `TCP_server.cs` is true the function `create_scene()` is called. After this function is finished the Boolean `newCrane` is set to `True`. Otherwise it is always set to `False`.

2.3.3 TCP_server.cs

This script is used to handle the data transfer between the python client and the unity server via a tcp socket connection.
The function `Start()` is called once at the start up of the application.
Inside this function a new thread is started. It will start the function `ListenForMessages()` to act as the `tcpListenerThread`. The last operation in the `Start()` function writes a string `"1"` in an external text file to tell the python script that the server is now running.

`ListenForMessages():`
This function starts with reading the file `server_tcp_config.json` and creates and starts a server as a `tcpListener` with the given address. After that the server tries to accept an client and if that works it will read of a stream from the client. The receiving data ends with the end tag `"eod."` and the data received before is set to the string `jsonparameters` which is then parsed to the object: `jsonCrane_here` of the class `JsonCrane`. If the stream ends with `"END.eod."` the server is stopped and `running_session = false` which leads to closing the whole application in the function `LateUpdate()`.

`LateUpdate()`
This function is called also once per frame and is executed after the `Update()` function. It tracks the time from the last time a picture was sent back to the python client or since the start up. If the timer reaches thirty seconds and no client is connected the `running_session = false` is set. If that is the case the application quits.
The last condition is entered if `newCrane=true` which is set in the script `create_crane.cs` and indicates that the scene is fully set up. Now the timer is set back to zero and an image is sent to the python client with the co routine `CapturePNGasBytes()`.

`CapturePNGasBytes():`
This function will make a screenshot with the requested resolution and send it as a byte array through a stream with an additional end tag to the python client.