Raytracer Service

> *“There is no such thing as shadow, it’s just the absence of light.”*

Kaihua Zhou 999482983

Ya Han Yang 1004550700

1. Introduction

Raytracer Service is a webapp providing 3D rendering with ray tracing algorithm. It aims to provide an easy-to-use demonstration of the algorithm, which can be handy for teaching and other purposes.

2. High Level Structure

Our app mainly has four functionalities:

- User Authentication
- Generate image with 3D objects (view & download)
- Set image ownership (public/private)
- Delete image from database

We use five components to achieve above functionalities:

- User Interface (front end webpage)

- Backend Logic
- Rendering Engine
- Database (DynamoDB tables)
- S3 Storage

Each of them are discussed in detail below.

2.1 User Interface

The webapp has these following pages:

Page	location	Functionality	Notes
Index Page	/index	Home page, allows user to login or register	Request to the root “/” will be redirected here
Login Page	/login	User enters credential to log into the system.	A failed attempt of login will bring up an error message
Register Page	/register	Allow user to create new account.	Username must be unique. After successful registeration, user will be redirected to their dashboard
Dashboard	/dashboard	A user’s personal page displaying user’s past rendering request (images in thumbnail) in descending order	An unauthenticated attempt to visit this page will be redirected to index page

		and allow new ones to be issued.	
Public Gallery	/public	Lists all public images, as thumbnails, in descending order.	Owner can set if a certain image is public. The image will be shown in this page and be accessible by all users.
Generate Request Form	/form	Create new rendering request. User must specify the objects information (coordinates, color, reflectivity, etc…) to render.	Currently allowed objects include sphere, triangle and rectangle. Maximum of 5 objects of each type (maximum of 15 objects in total). Redirect to dashboard page once the form is submitted.
Image Page	/image?image_id=[image_id]&page=[page]	It displays result of a certain rendering request. Result includes timestamp, ownership/owner, image and request entities. Owner is able to delete his/her image, change its ownership and download it on this page.	Redirected from dashboard or public gallery.

In case if an anonymous user tries to access any page except from the index, register, login page, he/she will redirected to the index page.

2.1.1 User Authentication

User can login or register with forms. When registering, we use javascript to first check whether the passwords has at least four characters and the passwords match. Username and password will be sent to backend as a POST request. In case if an error occured, user will see an error message above the forms.

2.1.2 Image Rendering and Viewing

To generate an image, an user has to go to the Generate Request Form page and fill in object information. Javascript will be called once user submits the form and will format object entities in json string. This json string will be send to backend as a POST request.

Image’s ownership is set to ‘private’ as default while rendering. Once the image is successfully generated, the owner will able to change its ownership on the Image page.

The generated images will be displayed as thumbnails on user’s dashboard. If an image’s status is still pending, it will be shown as a loading image in thumbnail. When clicking on this thumbnail, an message saying that ‘We are still generating the image.’ will pop up. The user is allowed to refresh the page or delete the image from the popup.

User will be allowed to view image in detail by clicking on its thumbnail. He/she will be redirected to an image page with the link “/image?image_id=[image_id]&page=[page]”. [page] is either ‘dashboard’ or ‘public’ defining whether the image comes from user’s dashboard or public gallery. This also defines where the ‘BACK’ button on the image page will redirect to. Modification buttons is only available if the image belongs to the current session user.

2.1.3 Image Ownership Updating

To modify the ownership of an image, the owner has to click on the ownership button on the image page. A request to “/updateOwnership/?image_id=[image_id]&page=[page]” will be sent. If the action succeeds, user will have a refreshed image page with the ownership property modified. Otherwise, user see an error message.

2.1.4 Image Deletion

Similar to ownership modification, a request to “/delete/?image_id=[image_id]&page=[page]” will be sent when owner clicks on the delete button. If the action succeeds, user will redirected to dashboard or public gallery page based on [page] value. Otherwise, user see an error message.

2.2 Backend Logic

The webapp is developed with Flask microframe work. It handles routing and processes form submitted by user. The backend also

interacts with the database and the rendering engine.

2.2.1 Functionalities

2.2.1.1 User Authentication

Upon registration, a “salt” (random string) is generated and then used to hash the password. The database only store salted and hashed password for security. When a user attempts to log in, the backend will query the database for stored salt and hashed password. Then password provided by user will be salted and hashed in the same way, before comparing with stored values.

2.2.1.2 Image Rendering and Viewing

When image rendering is requested, back-end will get image object entities from the POST request. It will randomly generate a image id (requestID), get current timestamp, then send a request to DynamoDB to insert the new request.

To view an image, the back-end will get the image id (requestID) and the page value. It will first request image data from the database then verify whether the image is public or belongs to session user.

If the user is allowed to see this image, the back-end will send all information that the front-end need to show. Otherwise, the back-end will render a template saying that the user is not allowed to see the image.

2.2.1.3 Image Ownership Updating

When an user requests to update the image ownership, the back-end first request image data from the database whether the image is public or belongs to session user. The new ownership will be set to the opposite of the previous ownership. It will render an error message template in case the user is restricted to make change to this image.

2.2.1.4 Image Deletion

Image deletion has similar process with updating image ownership. Please note that we only delete image records from the database. The actual images will be stored in S3 as long as the developers do not erase them from S3.

2.2.2 Database Connection

Since the backend is developed in Python, the standard boto3 library is used to interact with DynamoDB.

2.2.3 Invoking the rendering engine

The rendering engine is deployed as a separated lambda function (more on this later), and invoked using boto3 library.

Data about rendering request is sent in asyncronous “event” mode. The request will be set to “pending” state at first. Once the rendering is done, it will be set to “success” or “failed”.

2.2.4 Serverless Backend

The webapp can be deployed to AWS Lambda and API Gateway using a

library called “Zappa”.

2.3 Database

DynamoDB is used store user related information. Two tables are used. Their structure is shown below in json form.

Users Table

```
{
    "username": [str, hash key],
    "salt": [str,unique per user],
    "password": [str, salted and hashed]
}
```

Requests Table

```
{
    "requestID": [str, hash key],
    "stat":[str, can be either "pending", "success" or "f
ailed"],
    "ownership": [str, either "public" or "private"],
    "timestamp": [long int, formatted datetime, e.g. 2017
1125123010 for 2017-11-25 12:30:10],
    "username": [str],
    "entities": [str, json string detailing the objects o
f the rendering request]
```

```
}
```

2.3.1 Database Queries

We use Voto3 Python functions to send request to Dynamo DB from the backend.

INSERTION: Insert an image rendering request:

```
 req_table.put_item(Item=entry)
```

SCAN: Scan all requests belongs to the user:

```
 req_table.scan(FilterExpression=Attr("username").eq(user
name))
```

GET: Get the user info:

```
 user_table.get_item(Key={'username':username})
```

UPDATE: Update image ownership:

```
 req_table.update_item(
        Key={
            'requestID': image_id
        },
```

```
        UpdateExpression= "set ownership = :o",
        ExpressionAttributeValues={
            ":o": newOwnership
        }
    )
```

DELETION: delete an image record

```
  req_table.delete_item(
        Key={
            'requestID': image_id
        }
    )
```

2.4 Rendering Engine

For rendering, a python libaray called PyRT is used. It is a simple raytracer written in Python 3.6, making it a good option to deploy to AWS lambda. A wrapper libray was created to accept json styled input. Once the input is parsed, a 3D scene will be created then rendered into image. The image file is then uploaded to a S3 bucket for persistent storage.

We are aware that python is not the best language to implement raytracing (for Lambda, C# or Java may be better in performance). But since the rendering engine is a stand-alone Lambda function, one can build another (potentially better) engine and replace the current one,

as long as the following data interface is accepted.

More information on the PyRT library can be found here: https://github.com/martinchristen/pyRT

2.4.1 Rendering Data Interface

The request must be packed as a dictionary (in Python). It should conform the structure below:

```
{
    "id":[str, unique string representing the filename of
 rendered image, file extension must be included],
    "entities": [list, a list of dictionaries(json object
s) representing 3D objects, alongside with light and came
ra]
}
```

The entities can include any of the following:

Sphere

```
{
    "type":"sphere",
    "radius":"string representing a float number",
    "center":"x,y,z", #string of float numbers seperated
by commas
    "color":"r,g,b", #int  0-255 seperated by commas,
```

```
    "reflectivity":"string representing a float number 0-
1,"[optional]
}
```

Triangle

```
{
    "type":"triangle",
    "A":"x,y,z", #string of float numbers seperated by co
mmas,
    "B":"x,y,z", #string of float numbers seperated by co
mmas,
    "C":#string of float numbers seperated by commas,
    'color':'color string as in spheres',
    "reflectivity":"string representing a float number 0-
1,"[optional]
}
```

Rectangle

```
{
    "type":"rectangle",
    "A":"x,y,z", #string of float numbers seperated by co
mmas,
    "B":"x,y,z", #string of float numbers seperated by co
mmas,
    "C":#string of float numbers seperated by commas,
```

```
    'color':'color string as in spheres',
    "reflectivity":"string representing a float number 0-
1,"[optional]
}
```

Note that only 3 vertices are required to specify a rectangle. The fourth will be calculated automatically.

Camera

```
{
    "type":"camera",
    'position':"x,y,z", #location of the camera,
    'center':"x,y,z", # the location of center of the vie
w
    "width":[int or str representing an int],
    "height":[int or str representing an int] #width and
height of rendered image
}
```

Note: The camera will always set the “Up” vector to (0,0,1). The view vector is calculated by `center - position`. Only one camera is allowed. If multiple camera is specified, only the last one will be used. If no camera is provided, a default camera with `height=300, width=400, position=(0,-10,10), center=(0,0,0)` will be used.

Light

```
{
    "type": "light",
    "position":"x,y,z" #location of the light"
}
```

Note: Point light at given location. Same as the camera, only one light is allowed. If multiple ones are provided, only the last one is used. If no light is provided, a default one with `center=(0,0,15)` will be used.

3 Interesting Observations

3.1 Zappa Serverless Backend

Zappa helps packing WSGI web app built with flask, django and many others and deploy them to AWS Lambda. In theory, Zappa should work on all platforms. It appears to be so when deploying a simple “hello-world” type app with no extra libraries other than flask. However, during our attempt to deploy the Raytracer Service from a windows 8.1 machine, various errors occurred and some of them just cannot be fixed. The major one was that the deployed app was unable to import the “builtin” library in “handler” function, meaning that the app wouldn’t even start because of this import error.

The cause was (or at least we think so) that some the libraries include artifacts compiled from C. By default Zappa will deploy your app to an Linux-like environment and compiled C binaries are not cross-platform.

Instead, the deploying was eventually done using a Ubuntu virtual machine. Sources on the internet suggest an AMI linux EC2 instance would work as well. But even on Ubuntu errors were still encountered (yet fortunately they were the “fixable” ones).

Known issues are(as of the time writing this document):

1. Zappa cannot find a proper version for package “toml”, even when toml version 9.3.1 is installed
   Solution: use pip to install an older version `sudo pip install toml==9.2`
2. Zappa cannot work without a virtual environment.
   Solution: create a virtual environment with virtualenvwrapper. Then install required packages inside the virtual environment
3. Unable to deploy with Python3.6
   Work-around Solution: Since Zappa seems to have many issues with Python3.6, it would be safer to use older version of Python (Python3.5 or Python2.7) until those issues are fixed.

3.2 Deploying a function on AWS Lambda

AWS Lambda introduces an interesting new way of cloud computing. But when actually creating a function, there are many “gotchas” once your function goes beyond the basic hello-world. The official documentations, dare I say, are not very well written - vague, detail-omitting, “dummy-unfriendly”. And since Lambda is relatively new, the

community knowledge base is not really covering everything. Here are some of the issues we encountered when deploying the raytracing rending engine.

3.2.1 Only Prepare Deployment Package on Linux System

The official docs state that you have to include all libraries (except for boto3) in the deploy package. What they did not say was that you have to prepare your package using the same operating system (i.e. a linux system).

When I first tried to deploy using a windows machine, the lambda function failed on the first instant it was tested. The log showed that it could not load the core of Pillow library. Pillow, being a image processing library, has server C-compiled binaries. So the copy of Pillow I installed on the windows machine is uncompatible with Lambda which uses a Linux environment.

So always prepare your deploy package on Linux (virtual machine, EC2, etc.)

3.2.2 Remeber to Properly Set Time-out

Since our rendering engine is written in Python (meaning it’s not lightening fast), the first few tests of the Lambda function failed very quickly. After checking the logs, it was counted as timed out before rendering could finish. The default time-out was only serveral seconds. After increasing that to 120 seconds, everything went on smoothly.

3.2.3 Issues with Numpy on Lambda

While testing our rendering engine of Lambda, I found out that Lambda may have issues with numpy (version 1.1.13). By default, our render engine attempts to use Numpy array to store the pixles before dumping them into an image file. On serveral occations Lambda function logs showed that there was an error utilizing Numpy. I bypassed this by allowing the rendering engine to check if numpy is available and uses lists to store pixles when numpy cannot be imported. There are very little discussion about this on the internet, so we have to stick with this work-around for now.

Conclusion

Through this project, we have gained better understanding of dynamoDB, AWS Lambda and zappas. We've also developed skills in web development. If we have more time to improve the web application, we would probably ameliorate the UI design and add more avaiable types of image objects.