ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20170919221607

Original scientific paper

REMOTE RENDERING CONTROL USING PYTHON SCRIPTS AND DROPBOX TECHNOLOGY

Andrija BERNIK, Dinko GALETIC

Abstract: The process of rendering a 3D animation often takes a very long time to complete. In situations where it would take several hours or even many days, it is inconvenient to spend that time near the rendering computer in order to control and oversee the process. Through the work with 3D computer graphic technologies, the authors realized that there is no simple solution on the market that facilitates the monitoring of the remote computer that is running the rendering process. This paper deals with developing a system to enable those tasks to be done from a remote computer or any mobile device. The developed proof-of-concept system consists of two Python programs communicating over the Dropbox service and a computer that is running the Autodesk Maya software.

Keywords: Autodesk Maya; Dropbox; Python; remote rendering

1 INTRODUCTION

Rendering is a process in which a dynamic environment is stored as a 2D image. To create an image, it is necessary to use a system for calculating the position of a model and sources of light in a 3D space. The Autodesk Maya software solution was used in this paper for the purposes of 3D modeling and rendering. Since the process of rendering can take a very long time, sometimes even up to several hours for a single image [1] (usually called a frame), it is often inconvenient to be physically present by the rendering computer in order to oversee and control the process.

As for example, Jonah Lehrer said for the Wired Magazine regarding the making of Toy Story 3: "The average frame (a movie has 24 frames per second) takes about seven hours to render, although some can take nearly 39 hours of computing time. The Pixar building houses two massive render farms, each of which contains hundreds of servers running 24 hours a day." [1]

This paper addresses this problem by creating a system which enables the user to remotely control the process. The goal is to have a simple, portable system with as few dependencies as possible.

2 EXISTING RENDERING CONTROL SOFTWARE AND SETTING UP THE SCENE

The scene is created in Autodesk Maya and the focus is based on setting up the render preferences. Important to note is that there are two types of inputs for the complete setting up of the preferences and those are connected to the output file and quality of the rendering calculations.

Fig. 1 has some important options that directly influence the size and the type of rendering. Image format is set to jpg because of file size. The quality of an image is slightly lower but for the animation it is not a problem since we have 24 FPS (frames per second).

Frame extension is set to the "name.#.ext" format. This format is a must because of the Operating and file type (FAT / NTFS) system. [4]

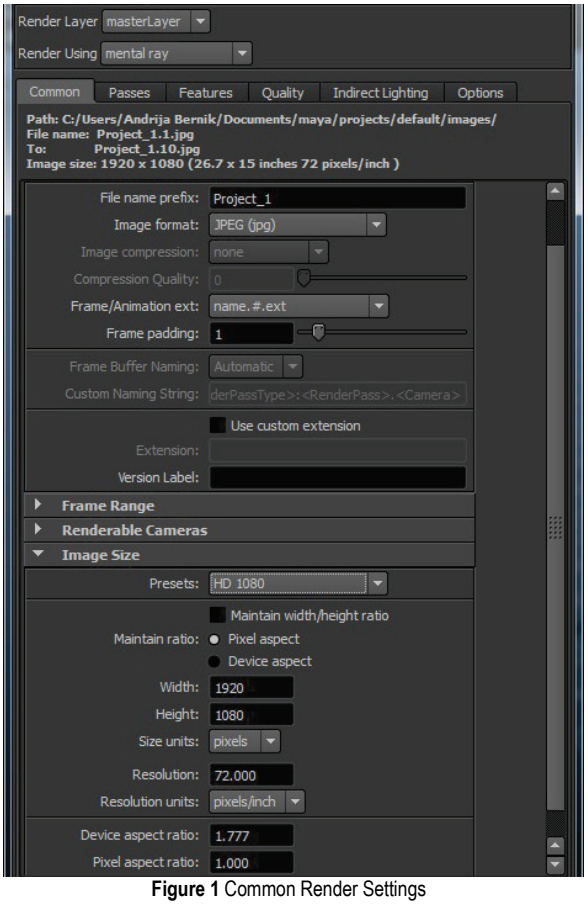

Figure 1 Common Render Settings

Rendered frames are meant to be used in the compositing software such as Adobe After Effects and that is why we also have to set the first and last frame for the rendering process. The last part of Common attributes is image size, which is determined by the use of animation. The other important feature is connected to the Quality itself. The user has the ability to choose from Quality presets if he or she does not have enough knowledge to

manually set the rendering stats. Rendering is not the main subject for this article which is why any further analysis will not be made.

There are several existing solutions for rendering available. The 3D Buzz's "Maya Renderer" application is created for the Windows Phone 7 mobile platform. This choice of platform imposes a limit on the application's usability: it cannot be used from a PC, and the Android and iOS platforms are more widely used when it comes to mobile platforms [2].

The "Mental Ray Satellite Tech" software can be used to control the rendering process through a Local Area Network (LAN), making it suitable for remote control only if Virtual Private Network (VPN) technology is used. Since such systems require some setting up and are often commercial, the authors did not find this solution appropriate.

The "Team Viewer" and similar screen casting applications give full control of the computer to a remote user by transmitting the controlled computer's entire screen over the network to the remote user's computer, and the remote user's mouse and keyboard input to the controlled computer. The authors find this solution inappropriate because its broad approach uses a lot of network bandwidth, while an application focused specifically on the control of the rendering process would use a significantly smaller amount. Such an application would be usable even in situations where bandwidth is severely limited, such as accessing Internet via a mobile phone while in roaming.

3 OVERVIEW OF THE DEVELOPED RENDERING CONTROL SYSTEM

The Python 2.7. Programming language was chosen to develop the solution. Python interpreters exist for a wide array of platforms [3], making the code written in Python almost entirely platform independent. Python's high-level approach is particularly suitable for solving a high-level problem such as this. Lastly, a large amount of Python modules [5] [6] greatly increases the functionality of the language.

The system consists of three parts:

a) The client application,
b) The server application,
c) The Dropbox server.

The client application runs on the computer from which one wishes to control the rendering process. The server application runs on the computer which does the rendering (referred to as 'the rendering server' in this paper). While the rendering computers are usually very strong computers, the client computer can be any computer capable of running Python 2.7. To allow communication between the client and the server computer, the Dropbox service was used. Dropbox is an online storage service which allows a user to store up to 2 GB of files for free and easily synchronizes them between different computers [7]. The Dropbox service can be used with a Dropbox application, through a browser or through their Application Programming Interface (API). The API is available in Python through the Dropbox's official python module.

In this solution, the rendering server uses the Dropbox application, and the client computer uses the Dropbox Python module in order to access the Dropbox server. This was used to show how both approaches can be used to allow such communication; alternatives where both computers use either the API or the application can similarly be made.

The server and client applications communicate by uploading and downloading files to and from each other via the Dropbox server. Those files are used to exchange commands for the server ("commands.txt"), to keep track of the Maya project currently being rendered ("active_rend.txt") and to keep track of all the projects available for rendering ("projects_to_rend.txt"). Additionally, the server uses a log file ("LOG.txt") to keep track of the commands it has already executed.

The server application does three basic tasks:

a) Check the commands file for updates,
b) Execute new commands,
c) Maintain the available project file.

Since a) and c) are input-output tasks that need to run continuously, each of the three tasks runs in its own thread of execution. Upon any updates to the command file, the commands are read and accordingly executed or queued for execution. If a new Maya project is added to the designated projects directory, the project name is added to the available project file.

The client application mostly runs in a single thread since it only needs to communicate with the Dropbox server upon the user's request and has no tasks it needs to run perpetually. It displays the data from the active rendering and available project files, and updates the command file when the user sends a new command. It briefly invokes a second thread to upload the updated command file in order to avoid the application from blocking while the file is being uploaded.

4 DROPBOX API SPECIFICS

In order for an application to be able to use the Dropbox API, it needs to be registered on the Dropbox website, where it receives two pieces of information unique to it, called "APP_KEY" and "APP_SECRET". The creator of the application has to choose whether the application will have access only to a single directory in the user's Dropbox account, called the "Apps" directory, or to everything in the account. The latter requires a special permission from the Dropbox staff, but the former was sufficient for this application. The key, the secret and the access type information are required to use the API.

The user of the application allows the application to connect to their Dropbox account by following a link to the Dropbox website (called the 'authorization url') specially generated by the application and agreeing to permit the application to connect to their account. Once the application is permitted, the Dropbox server returns the "access token", which is an object containing the credentials signifying the

permission. The authorization can be remembered for future uses by saving the token to a hard drive and loading it when needed. [11] The required parts of the Dropbox module can be imported with the following:

```
from dropbox import client, rest, session
```

In order to start a Dropbox session, the application needs to run the following code:

```
sess = session.DropboxSession (APP_KEY,
APP_SECRET, ACCESS_TYPE)
```

In order to create the authorization url, the application must first obtain a "request token" from the Dropbox server and use it to create the url:

```
request_token = sess.obtain_request_token()
url = sess.build_authorize_url
(request_token)
```

The url thus obtained is a plain text string. Once the user follows it and allows the application to access their Dropbox account, the following line of code is used to fetch the authorization credentials from the Dropbox server:

```
sess.obtain_access_token (request_token)
```

The credentials are stored in the `sess` object as the **`sess.key`** and **`sess.secret`** variables. Note that these are unrelated to the before mentioned APP_KEY and APP_SECRET variables. Finally, the **`sess.set_token`** method needs to be called with the **`key`** and **`secret`** variables before we can use the rest of the API.

```
key, secret = sess.token.key,
sess.token.secret
sess.set_token(key, secret)
```

If the access token was saved to the hard drive, only the last step is required. One would load the `key` and secret variables from a file and call the **`set_token`** as shown above.

Uploading a file to the Dropbox server and downloading a file from it can each be accomplished by one line of code. First, a "client" object is required.

```
klijent = client.DropboxClient(sess)
```

The **`client.get_file`** method returns an open file object which can be read or saved as a file object returned by the Python's built-in **`open function.`** The **`client.get_file`**.

The method requires an argument specifying the path to the file in the user's account, with the root directory being the one the application is allowed to use. The Dropbox API uses the Unix notation of forward slashes ("/") for paths.

```
dat = klijent.get_file ("/path/to/file.txt")
```

To upload a file, the **`client.put_file`** method needs to be called with the remote path (the name and the location under which to save the file) and an open local file object parameters specified. By default, the file will be automatically renamed to avoid overwriting an existing file with the same name. The function returns an object containing the uploaded file's metadata. This operation is synchronous and will block the program until the file is uploaded.

```
response = klijent.put_file (remotepath,
local_file)
```

5 IMPLEMENTATION DETAILS FOR THE CLIENT SIDE APPLICATION

The rendering process can be invoked by using a command line interface (CLI) program called "render" without starting the Maya Software. [8] The parameters passed to "render" can be used to set various rendering details. This approach was very suitable for this solution because sending, receiving and processing a line of text can easily be done and requires very little network bandwidth. This solution allows the user to choose which rendering algorithm to use and to set which frames are to be rendered by choosing the first and the last frame. The necessary variables for the CLI command are determined by the user's choices and added to the command file, after which a new thread is started to upload the file. [9]

```
write_me = "rend:render -r " + render + " -s
" + start_frame +  end_frame + " " +
project_file + "\n"
dat = open(COMMANDS_FILE, "a")
dat.write(write_me)
dat.close()
Thread(target = self.send_work).start()
```

The **`Thread`** object comes from the Python **`threading`** module. The **`send_work`** method wraps around a generic uploading function and calls it to upload a specific file. While this is the only instance in the application where the generic function is used, that function was written with future expansions in mind.

```
def send_work(self):
    """ Upload the commands file to the
Dropbox server."""
    self.__upload__(COMMANDS_FILE,
remotepath = "/Maya/commands.txt", overwrite
= True)
```

In the **`__upload__`** method, the API **`put_file`** method is called in the following way. The `overwrite` parameter is set to `True` because the API does not offer a way to append to a file, making it necessary to overwrite the existing one. The **`expanduser`** method provides a portable way to use file names.

```
local_file = open(os.path.expanduser
(localpath))
response = self.klijent.put_file
(remotepath, local_file, overwrite)
```

Besides the command to initiate the rendering process, the application supports the "progress" command. The command requests the server to upload the latest rendered image to Dropbox so the user can check how far the rendering process got and whether the scene looks as expected.

The list of available tasks is downloaded by the **fetch_tasks** method. A similar method called **fetch_active_rend** does the same thing, with the only difference of opening a different file and saving in a different variable.

```
def fetch_tasks(self):
    """ Download a list of projects awaiting
rendering."""
    dat = self.klijent.get_file
("/Maya/projects_to_rend.txt")
    try:
       data =  dat.read()
        # Each element of "data" is
       a string representing a project file
       (*.mb or *.ma).
        data = data.rstrip().split("\n")
    except:
        data = []
    self.task_list = data
```

6 IMPLEMENTATION DETAILS FOR THE SERVER SIDE APPLICATION

Once started, the server side application asks the user to enter the paths to the required directories and files: the Dropbox directory the application is allowed to use, the directory where the Maya projects are stored, the path to the directory where the rendered images are stored and the path to the log file. Afterwards, the application initiates the required threads and runs in the background, requiring no further user input. The main thread continuously checks the command file for updates. A portable way to do that is to keep checking the file modification time, stored in **the st_mtime** variable.

```
old_modify_time = os.stat(dropbox_path +
"commands.txt").st_mtime
while(True):
    try:
        new_modify_time =
os.stat(dropbox_path +
"commands.txt").st_mtime
    except:
        # The exception will happen
if the file is beign written to while
attempting to read it;
        # waiting 15 seconds should be
sufficient for the writing to finish.
        time.sleep(15)
        new_modify_time =
os.stat(dropbox_path +
"commands.txt").st_mtime
    if new_modify_time != old_modify_time:
        old_modify_time = new_modify_time
        dat = open(dropbox_path +
"commands.txt", "r")
        commands =
dat.read().split("\n")
        dat.close()
```

Since the command file contains every command that was ever sent to the rendering server, a log is maintained to keep track of what commands were executed. The new commands are now read and processed. The lines starting with "rend:" represent the commands to render a project, and those commands are appended to a queue variable, to be read by the rendering thread. The lines saying "progress:" are requests for progress updates. Since these are just a matter of finding and copying the latest image to the Dropbox directory, they can be executed quickly and thus do not require a specific thread to execute them.

```
for com in commands_to_run:
    last_command_number += 1
    if com[0:4] == "rend":
        q.append(com[5:])
    elif com[0:8] == "progress":
        progress()
```

The rendering thread runs the following code. Since the **subprocess.call** blocks the calling thread until the sub process finishes, the thread will be blocked until the rendering process is complete.

```
def render_thread():
    """ Perpetually checks the queue.
Executes commands one by one."""
    while True:
        while len(q) > 0:
            command = q.pop(0)
            print command
            dat = open(dropbox_path +
"active_rend.txt", "w")
             # Write the name of the
             file being rendered.
          dat.write(command.split()[-1])
            dat.close()
            # Run the command as the
             CLI command.
            result =
subprocess.call(command.split())
             # Clear the active_rend.txt
             file.
            dat = open(dropbox_path +
"active_rend.txt", "w")
            dat.close()
        time.sleep(5)
```

7 PROBLEMS ENCOUNTERED DURING DEVELOPMENT

The Dropbox API documentation [10] contained an error regarding the **obtain_access_token** method at

the time of writing of this paper. The method does not return a tuple object containing the key and the secret as the documentation states; it stores the token in the session variable.

```
# key, secret =
sess.obtain_access_token(request_token) #
Error!
sess.obtain_access_token(self.request_token)
print sess.token.key, sess.token.secret #
Correct.
```

Various bugs occurred due to the differences in how the GNU/Linux and Windows operating systems treat text files and file paths. GNU/Linux separates the lines in text files using the line feed (LF) character, while Windows uses both the carriage return (CR) and the LF characters, thus separating the lines with CRLF (depicted in Python as "\r\n"). Both the server and the client application expect the lines to be separated with just the LF. While this works fine, Windows text editing applications may show all the text from this solution's files as a single line. The difference in how the OSs treat file paths (Windows separates directories with a backslash ("\"), and GNU/Linux with a slash ("/")) were avoided by using the `os.path.expanduser` method instead of using file paths directly.

8 FUTURE DEVELOPMENT POSSIBILITIES

As a proof-of-concept application, it contains a very basic set of just two commands that can be passed to the rendering server. It can easily be expanded to include other commands, such as stopping the rendering process or changing the priority of the queued tasks. Advanced options such as allowing multiple machines per user or controlling an entire rendering cluster could be added as well.

Concurrency issues are possible if a thread attempts to read its file while the file is not done downloading. This was avoided by using a try-except block which delays the reading if an error occurs, but a proper solution would be to use semaphores and possibly the file metadata to determine when the downloading was finished.

Semaphores could also be used to keep the rendering thread inactive when there are no tasks waiting instead of using `time.sleep(5)` to have it checked for new tasks every five seconds.

The application could be expanded to support more platforms. Mobile phones would be particularly suited for the purpose of remotely controlling the rendering computer. Finally, support for a 3D rendering software besides Maya could be added.

9 CONCLUSION

This article shows that even though there is not a single PC solution that enables the user to remotely control Maya's batch render, authors have successfully combined the given technology. Dropbox is essential and required for transferring and triggering python scripts which then communicate with Maya's batch render in the background process of the Windows platform. The problems were based on the differences between the two operating systems. Moreover, there is a flaw inside the Dropbox documentation which needs to be corrected for future user readings.

If running the code samples from the article, please note that the article formatting often broke longer lines into two or more lines and should be adjusted accordingly.

Maya's CLI interface is suitable for use by another program to control the rendering process. A rudimentary application which remotely controls the process can be developed in a few days and there is much potential for expanding its functionality.

10 REFERENCES

[1] Lehrer, J. (2010). Animating a Blockbuster: How Pixar Built Toy Story 3, http://www.wired.com/magazine/2010/05/process_pixar/all/ (Accessed: 06.04. 2017).

[2] van der Meulen, R. & Rivera, J. (2013). Gartner Says Worldwide Mobile Phone Sales Declined 1.7 Percent in 2012, http://www.gartner.com/newsroom/id/2335616 (Accessed: 10.04.2017).

[3] Python Software Foundation: Download Python for Other Platforms, http://www.python.org/getit/other/, Accessed 11.04.2017.

[4] Preda, M., Villegas, P., Morán, F., Lafruit, G., & Berretty, R. P. (2008). A model for adapting 3D graphics based on scalable coding, real-time simplification and remote rendering. *The Visual Computer*, 1, 24(10), 881-888. https://doi.org/10.1007/s00371-008-0284-2

[5] The Python Standard Library, Python Software Foundation, 2013, http://docs.python.org/2/library/ (Accessed: 13.04.2017).

[6] PyPI - the Python Package Index, Python Software Foundation, 2013, https://pypi.python.org/pypi (Accessed: 13.04.2017).

[7] Dropbox, Inc.: Dropbox - Tour, https://www.dropbox.com/tour (Accessed: 11.04.2017).

[8] Palamar, T. & Keller, E. (2011). *Mastering Autodesk Maya 2012*. Wiley Publishing, Inc. Indianopolis, Indiana, 657-664.

[9] Autodesk Maya: Maya User's Guide: Start Maya from the command line, available at http://download.autodesk.com/global/docs/maya2013/en_us/index.html?url=files/Interface_overview_Start_Maya_from_the_command_line_.htm,topicNumber=d30e12859, Accessed 07.02.2017.

[10] Dropbox Inc.: Dropbox Client - Dropbox Documentation, https://www.dropbox.com/static/developers/ dropbox-python-sdk-1.5.1-docs/index.html#dropbox.session.DropboxSession.obtain_access_token (Accessed: 06.02.2017).

[11] Drago, I., Mellia, M., M Munafo, M., Sperotto, A., Sadre, R., & Pras, A. (2012). Inside Dropbox: understanding personal cloud storage services. *Proceedings of the 2012 ACM conference on Internet measurement conference*, 481-494. https://doi.org/10.1145/2398776.2398827

Authors' contacts:

Andrija BERNIK, Ph.D.
University North,
42000 Varazdin, Croatia
andrija.bernik@unin.hr

Dinko GALETIC, Mag. inf.
Self-employed programmer,
47000 Karlovac, Croatia
dinko.galetic@gmail.com