MICRORAPTOR GUI - A LIGHTWEIGHT REMOTE RENDERING PROCESS MONITORING SOFTWARE

Ivan Radosavljević*,
Mladen Vidović,
Nebojša Nešić

Singidunum University,
Belgrade, Serbia

Abstract:
The rendering of a large number of images is a demanding task, usually delegated to remote rendering farms. This necessitates the creation of software for the management of these remote rendering tasks. Several commercial and non-commercial solutions can be used for this task. However, they are not specialized for rendering, and thus require either additional configuration and expansion, or familiarity with the remote rendering machines and the manual setup of rendering tasks. In this paper we present a solution to alleviate this issue. The solution consists of two components: Microraptor GUI and Microraptor client. Microraptor GUI is a lightweight web-based task monitoring and manipulation panel. Its backend is implemented in Python using the Flask framework and the frontend is implemented using the Angular framework. Microraptor client is a Blender plugin that allows for the direct access to rendering processes.

Keywords:
Remote rendering, Remote control software, Distributed rendering.

INTRODUCTION

Rendering a large number of images can be a computation-heavy task, especially in the case of photorealistic, high-resolution images. Thus, these tasks are usually performed on dedicated hardware or clusters of dedicated hardware, often called render farms. Such hardware is usually placed in rooms specially designed to house it, protected from moisture and with efficient cooling systems as it is often required to operate under high load for long periods of time. This makes the hardware inaccessible to its end users without specialized software tools for remote access. The most utilized solution for this is the use of a Secure Shell (SSH) client. As these clients often provide only a command line interface, they require the users to be sufficiently proficient and familiar with terminal syntax and commands available on the system they are accessing. The command line interface also limits the use cases for such applications, as they do not provide means to visually inspect rendered images or track the rendering progress and hardware load without the use of specially written scripts that output that information to the console.

Correspondence:
Ivan Radosavljević

e-mail:
iradosavljevic@singidunum.ac.rs

DOI: 10.15308/Sinteza-2021-141-145

Another solution is the utilization of screen sharing and remote control software. This includes commercial solutions such as TeamViewer [1] or AnyDesk [2], as well as open-source software such as Apache Guacamole [3] and Remmina [4]. These tools function by streaming the screen of a host machine to a client, as well as sending commands from the client to the host.

This allows users to access the desktop of a remote machine and manipulate it through a GUI as they would a local machine. Although easier to an end user, this process is more resource demanding than plain SSH access. These tools are meant as a general-purpose remote access solution, mostly for technical support. As such, they are not specialized for the monitoring and control of rendering tasks. Web-based tools for remote system administration such as cPanel [5] and Webmin [6] provide a more specialized environment with tools for remote system configuration. These solutions provide tools to track system load, start, end and monitor processes, access and manipulate the host file system through a web client. As these tools are often operating system agnostic, the end user does not need to be familiar with the remote machine's system, only with the tool they are using. However, these solutions do not provide native tools specialized for the manipulation of rendering processes.

In this paper, we present Microraptor GUI, a distributed remote rendering process monitoring software with a web-based GUI. In the second chapter, we provide an overview of related work. In III, the software architecture is presented. The fourth chapter contains a discussion of implementation details. In V, we demonstrate our solution using a simple use case scenario. Finally, in VI, we conclude with an overview of the implemented solution, and a discussion of possible future improvements.

2. RELATED WORK

In [7], the authors present a solution for a grid-based rendering farm based on Condor, an open-source high throughput computing software framework. A configuration file, containing rendering parameters is passed to the Condor software via a command line interface. The authors note that a major drawback of this approach is the difficulty of manually writing the configuration files and deploying them to the grid. In a follow-up paper [8], the authors attempt to solve that issue by implementing a GUI for the generation of the aforementioned script files.

The created interface also allows for the tracking of currently running jobs. Checkpoints for individual jobs are not supported. Hence, if a job fails during rendering, the entire job must be restarted. Due to this deficiency, the authors suggest the creation of smaller individual jobs.

In [9], the authors present a collaborative animation rendering system in order to speed up the rendering of digital animated videos created by undergraduate art students. However, the system provided only a command line interface for the remote control and monitoring of tasks.

As such, it required its users to be familiar with cluster computing and Linux terminal commands, which presented a barrier for most students. In order to increase the usability of the rendering system, the authors expanded upon their work in [10] by creating a simpler GUI in the form of a web portal. The portal supports user registration and authentication. Users can upload Blender files which contain the 3D animations to be rendered and submit rendering jobs. An overview of previously submitted and currently running jobs, as well as a preview of rendered videos are also provided. The interface lacks advanced output settings, allowing only for the generation of a single video from a Blender file. This is due to the GUI being designed for a narrow use case.

3. SOFTWARE ARCHITECTURE

Our proposed solution is a distributed rendering system. A central node is utilized as an access point for end users, which can be used to assign new tasks and monitor running tasks. Tasks are distributed to Microraptor clients which are installed on individual rendering nodes. Each rendering node executes a single rendering task and returns metadata associated with the task, and the URLs of the render output files. The metadata contains the rendering node's hardware load, which includes CPU, GPU and RAM load, global rendering parameters, as well as the progress of the currently running rendering task. Figure 1 shows the data flow and main components of the Microraptor GUI rendering system.

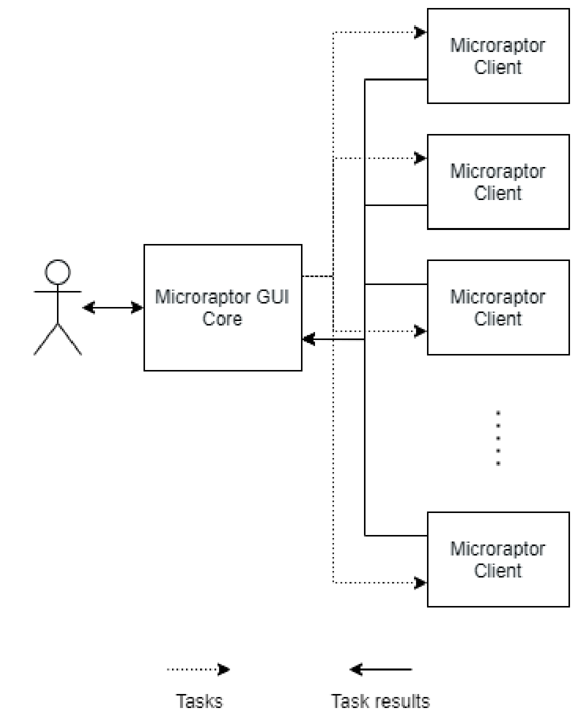


Figure 1 - Architecture overview

Figure 2 shows the class diagram of our proposed solution. Each user can create an arbitrary number of tasks, by assigning them a title, start and end frames if the rendering task in question is an animation, and a priority, the value of which will determine the task's position in an execution queue. For each rendering output of a task, a task result is generated that contains the URL of the output file, the date of task termination, as well as a status message. Additionally, each task result may contain an arbitrary number of TaskResultMetadata objects. These objects contain key-value pairs, allowing for a flexible definition of a task result's metadata.

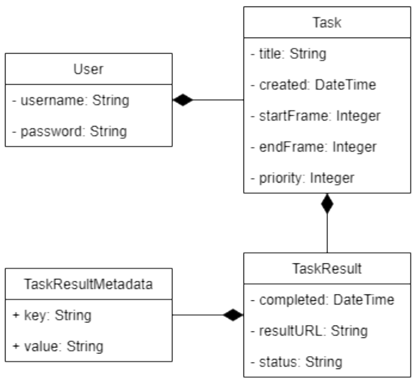


Figure 2 - Class diagram of the Microraptor GUI application

A sequence diagram illustrating a typical use case is shown in Fig 3. A logged in user creates a task by accessing Microraptor GUI. The user is then notified about the successful creation of the task. Once the user commences task execution, the task is deployed to a Microraptor client for rendering. Upon deployment of the task, the Microraptor client confirms the task deployment to the Microraptor GUI, which then informs the user that the task was successfully deployed. While the task is being executed, task results are sent from the Microraptor client to Microraptor GUI and accumulated there. The user can request a status update of the task being executed, upon which all the accumulated task results are returned to the user by the Microraptor GUI.

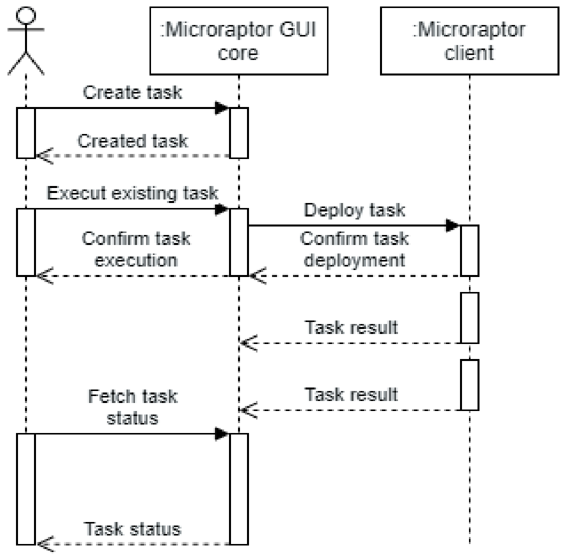


Figure 3 - Sequence diagram of a typical usecase

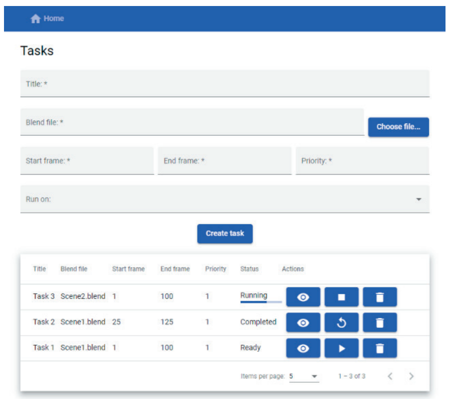
4. IMPLEMENTATION DETAILS

The backend of Microraptor GUI is implemented as a web application in the Python programming language, using Flask, a lightweight WSGI framework [11]. The frontend of the application is implemented using the Angular framework [12] and the Angular Material UI component library [13]. Apache CouchDB [14] is used to store the task results received from the clients.

The Microraptor client is implemented in Python as a Blender plugin. This approach allowed us to directly access the rendering process and extract data such as the current frame being rendered, estimated render time and render result. The Python os and psutil libraries were used to extract system information such as CPU, RAM and GPU load.

The communication between the Microraptor GUI and the Microraptor client is realized by using the ZeroMQ [15] library.

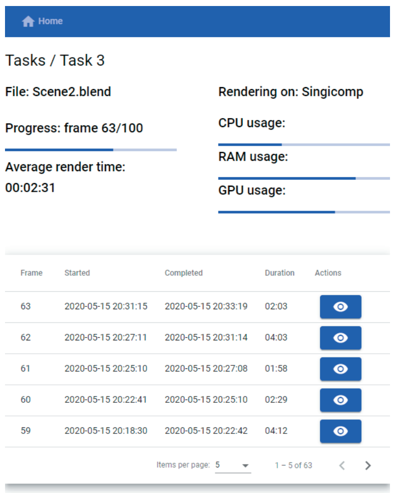
Figure 4 shows the task overview panel of the Microraptor GUI. It contains a form for adding a new task, and a table displaying previously added tasks. For each task, the user can view its current status, request additional details, deploy it, terminate current deployment or restart a completed task.



Title	Blend file	Start frame	End frame	Priority	Status	Actions
Task 3	Scene2.blend	1	100	1	Running	
Task 2	Scene1.blend	25	125	1	Completed	
Task 1	Scene1.blend	1	100	1	Ready	

Figure 4 - Tasks overview panel

Figure 5 shows the task details overview for a single task. On this page, the user is presented with the name of the current scene being rendered, the name of the node it is rendered on, current rendering progress and average rendering time per frame. This page also contains information about the hardware load of the node. Each rendered frame is displayed in a table and can be viewed independently.



Tasks / Task 3

File: Scene2.blend

Rendering on: Singicomp

Progress: frame 63/100

CPU usage:

RAM usage:

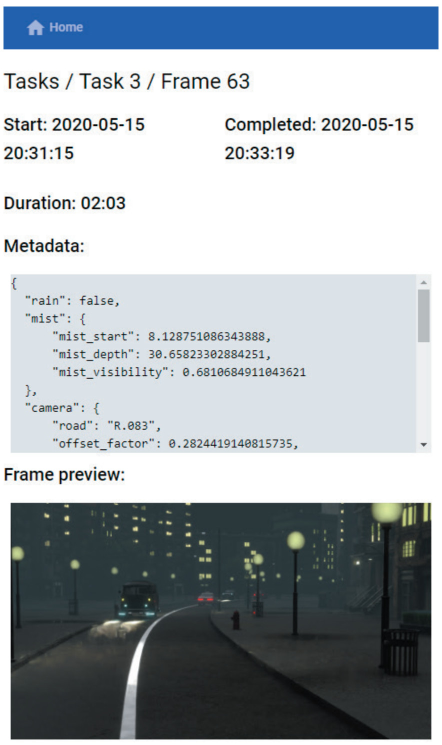
Average render time: 00:02:31

GPU usage:

Frame	Started	Completed	Duration	Actions
63	2020-05-15 20:31:15	2020-05-15 20:33:19	02:03	
62	2020-05-15 20:27:11	2020-05-15 20:31:14	04:03	
61	2020-05-15 20:25:10	2020-05-15 20:27:08	01:58	
60	2020-05-15 20:22:41	2020-05-15 20:25:10	02:29	
59	2020-05-15 20:18:30	2020-05-15 20:22:42	04:12	

Figure 5 - Task details panel

Figure 6 is the preview of an individual frame. It contains the start and end time, and duration of the rendering process, as well as metadata associated with the frame, and an image preview.



Tasks / Task 3 / Frame 63

Start: 2020-05-15 20:31:15

Completed: 2020-05-15 20:33:19

Duration: 02:03

Metadata:

```
{
  "rain": false,
  "mist": {
      "mist_start": 8.128751086343888,
      "mist_depth": 30.65823302884251,
      "mist_visibility": 0.6810684911043621
  },
  "camera": {
      "road": "R.083",
      "offset_factor": 0.2824419140815735,
```

Frame preview:

Figure 6 - Frame details panel

5. CONCLUSION

In this paper we showed a solution for monitoring and controlling a remote rendering process using a lightweight distributed web-based software. The software consists of two distinct components: Microraptor GUI, an end user access point for the software, and Microraptor client, a Blender plugin that enables direct access to the rendering process. The solution allows for the management of rendering tasks using a GUI, thus eliminating the need for familiarity with the rendering node's system and command line tools. The proposed solution can be expanded upon in order to facilitate the deployment of tasks other than the rendering of a scene. Another direction for improvement would be to update task results on the Microraptor GUI immediately upon a change on the client, without the end user needing to explicitly request updates.

REFERENCES

[1] "TeamViewer - The Remote Connectivity Software," 6 2021. [Online]. Available: https://www.teamviewer.com/en/.

[2] "AnyDesk," 6 2021. [Online]. Available: https://anydesk.com/en.

[3] "Apache Guacamole™," 6 2021. [Online]. Available: https://guacamole.apache.org/.

[4] "Remote desktop client with RDP, SSH, SPICE, and VNC protocol support. - Remmina," 6 2021. [Online]. Available: https://remmina.org/.

[5] "cPanel," 6 2021. [Online]. Available: https://cpanel.net/.

[6] "Webmin," 6 2021. [Online]. Available: https://www.webmin.com/.

[7] Z. Patoli, M. Gkion, A. Al-Barakati, W. Zhang, P. Newbury and M. White, *How to Build an Open Source Render Farm Based on Desktop Grid Computing,* 2008, pp. 268-278.

[8] M. Z. Patoli, M. Gkion, A. Al-Barakati, W. Zhang, P. Newbury and M. White, *An open source Grid based render farm for Blender 3D*, 2009.

[9] T. Boettcher and N. Wolf, "DSABR : Di stributed System for Automated Blender Rendering," 5 2013.

[10] J. Rankin, T. Boettcher and P. Bui, A Web Portal For An Animation Render Farm.

[11] "Welcome to Flask – Flask Documentation (2.0.x)," 6 2021. [Online]. Available: https://flask.palletsprojects.com/en/2.0.x/.

[12] "Angular," 6 2021. [Online]. Available: https://angular.io/.

[13] A. Material, "Angular Material," 6 2021. [Online]. Available: https://material.angular.io/.

[14] "Apache CouchDB," 6 2021. [Online]. Available: https://couchdb.apache.org/.

[15] "ZeroMQ," [Online]. Available: https://zeromq.org/.