Cloud-Based Ray Tracing Animation Generator

Daniel Shu
Vanderbilt University
daniel.m.shu@vanderbilt.edu

Xinyu Niu
Vanderbilt University
xinyu.niu@vanderbilt.edu

Yueqi Li
Vanderbilt University
yueqi.li@vanderbilt.edu

***Abstract*—Ray tracing is a graphics rendering technique that is powerful but has historically been extremely time and computationally expensive. Until very recent times, other techniques have been more popular for applications that require quick responses (such as real-time graphics). Cloud technologies have the capability to provide us the resources to overcome some of the speed limitations of traditional ray tracing. We have developed an application that utilizes this combination of technologies to quickly and accurately generate animated images.**

***Index Terms*—ray tracing, parallel processing, Kubernetes**

I. Introduction

A. Overview

The animation generator we designed is intended to provide an efficient, light-weighted, cloud-based service that creates a ray traced animation based on the user's specification. The application employs a C++ ray tracing program which takes in geometric and physical light properties as parameters to render a 3D digital image. One highlight of our application is the full use of cloud computing which conserves computational resources and time consumption on the user-side. Once a request is sent to the application, frames are concurrently rendered within the cloud before being compiled into a GIF animation and saved on the user-side device.

B. Ray Tracing

Ray tracing is a graphics rendering technique that creates photorealistic images based on a simulation of light transport. It calculates path of each light beam as it encounters objects in a 3D scene, and simulates a wide range of optical effects, such as reflection, refraction, shadow, ambience, depth of field and dispersion phenomena [1]. The ray tracing technique has a revolutionary effect on certain types of design where light path simulation facilitates physically accurate design, such as mirror and jewelry. Designers use ray tracing to create a better representation of the product for evaluation and at the final stage of advertisement, marketing staff use ray tracing to render realistic digital images and display them to consumers [2]. The technique is also widely used in 3D animation and film since lighting artists could create high-quality light simulation and camera effect at a low cost. The Walt Disney Animation studios posted an article about the significant role of ray tracer in the animation movie 'Big Hero 6'. The technique allows displaying surprisingly realistic 3D scenes in an efficient way, eliminating the need to use tricks like painting to show distant scenery [3] [4].

However, the biggest challenges for promoting the adoption of ray tracing technology in real-time applications, such as video games, are its heavy computational requirements and slow computation process [5]. Unlike films and animation, video games require extremely high responsiveness. If the ray tracer cannot provide rapid rendering of frames based on real-time inputs, the game experience would be compromised. Similarly, if it takes up too much disk space or has high hardware requirements, users are less motivated to install the game. Thus, speeding up the ray tracing model and reducing computational resources for users are indispensable for future development.

C. Motivation

Due to the slow processing and highly computational requirement of the ray tracing technique, creating animated images would be more time-consuming and challenging for users who don't have advanced hardware support. Cloud computing provides one possible solution to these drawbacks. We saw an opportunity to combine our experiences from the graphics and cloud computing courses and work on an application that has the potential to be applied to various industries after some refinement. For example, this cloud-based rendering approach could provide many benefits to video game production [6] since the improved speed ensures a more smooth game experience and helps eliminate disk space limitations.

II. Design and Architecture

As displayed in Fig. 1, our application is cloud-based with containerized services in two nodes managed by Kubernetes with one hosting utility services and one hosting rendering services. The services in the utility node are Kafka Zookeeper, two Kafka brokers, and a GIF compiler that puts together a collection of image frames into a GIF file format. The renderer node has greater RAM and CPU capacities and hosts a scalable number of ray tracers that render images based on input scripts.

A. Nodes and Scalability

The main purpose of Kubernetes in our architecture is to provide quick and efficient scaling of our ray tracers as pods. We split the services into the utility and renderer nodes in order to organize the services according to their tasks. Furthermore, we could also scale the amount of renderer nodes in an organized fashion. As for the difference in the resources available

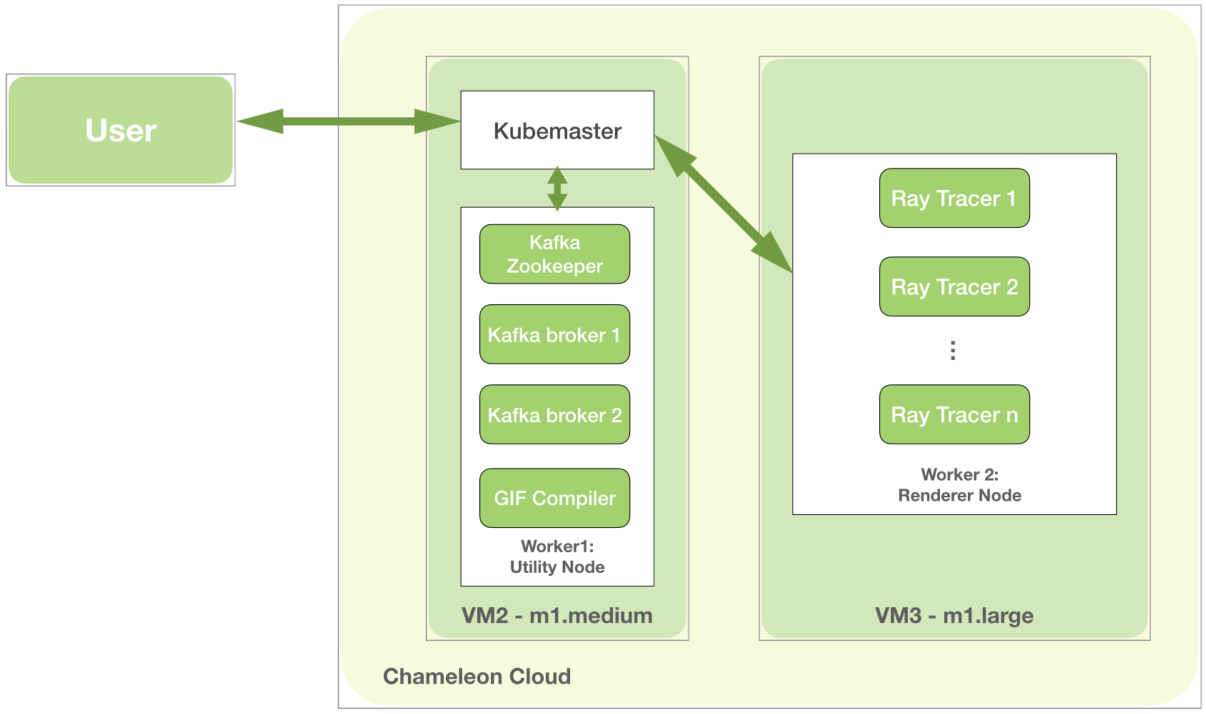


Fig. 1. Architecture Model

to each node, the renderer node was given more powerful specifications because the more computationally heavy tasks are processed within it.

B. GIF Compilation

We chose to containerize the GIF compilation step rather than execute it locally on the user-side because this further minimizes the computational requirements of the user device. Unfortunately, we encountered an implementation problem that is detailed in Section III-B5.

III. Implementation

A. Technologies

The technologies we used are Chameleon Cloud, Ansible, Docker, Kubernetes, Apache Kafka, a custom OpenGL ray tracing program, and various specialized Python libraries.

1) Chameleon Cloud: Chameleon served as our cloud platform. We chose it because it we had been using it throughout the semester and are more familiar with it compared to alternatives.

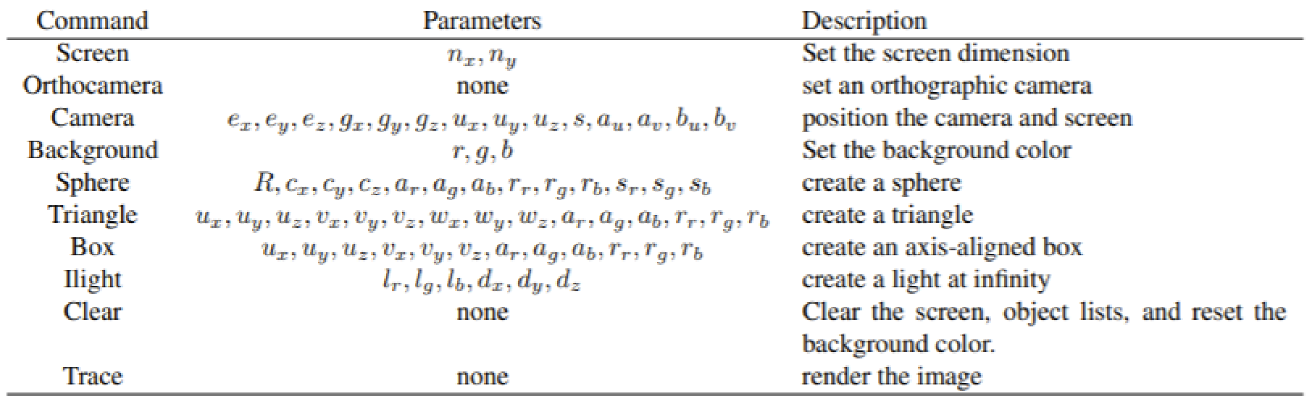
2) Docker: Docker provided us the functionality to containerize all of our services. For each service, a dockerfile was written to build a barebone Ubuntu-based image with just the packages and files necessary to run the service.

3) Kubernetes: Kubernetes then allowed us to manage a distributed system of containers across different virtual machines. We wrote YAML files to configure the workload resources necessary for setting up our services in pods. For Kafka ZooKeeper and the Kafka brokers, we set up Kubernetes Services to allow them communication with the other pods and, in the case of the brokers, programs outside the cluster. All of our pods were created using Deployments. The number of pods holding ray tracers was made scalable using the ”Replicas” specification of Deployments.

4) Ansible: We used Ansible to automate the processes of setting up virtual machines in the cloud, installing Kubernetes and Docker, setting up the Kubernetes cluster, adding the nodes, building the images, and running the Kubernetes workload resources. This allowed us to efficiently set up and run all the necessary infrastructure in the cloud. Furthermore, it provides us the ability to easily and efficiently scale the number of nodes if ever necessary.

5) Apache Kafka: Apache Kafka was used for all message passing between the different services within the cluster as well as between the services and the user code. We had 2 brokers being managed by the ZooKeeper service. Configuring the number of partitions in the brokers allowed us to increase or decrease the parallelism of the ray tracing because all ray tracer consumers were in the same consumer group. For our test runs, we matched the number of partitions to the number of ray tracer pods (5 or 10).

6) Custom Ray Tracing Program: For our ray tracing program, we reused the C++ code we had previously written for the graphics course at Vanderbilt University. The algorithmic logic of ray tracing was implemented manually by us, and the rendering process was done using the OpenGL API.



Command	Parameters	Description
Screen	n_x, n_y	Set the screen dimension
Orthocamera	none	set an orthographic camera
Camera	$e_x, e_y, e_z, g_x, g_y, g_z, u_x, u_y, u_z, s, a_u, a_v, b_u, b_v$	position the camera and screen
Background	r, g, b	Set the background color
Sphere	$R, c_x, c_y, c_z, a_r, a_g, a_b, r_r, r_g, r_b, s_r, s_g, s_b$	create a sphere
Triangle	$u_x, u_y, u_z, v_x, v_y, v_z, w_x, w_y, w_z, a_r, a_g, a_b, r_r, r_g, r_b$	create a triangle
Box	$u_x, u_y, u_z, v_x, v_y, v_z, a_r, a_g, a_b, r_r, r_g, r_b$	create an axis-aligned box
Ilight	$l_r, l_g, l_b, d_x, d_y, d_z$	create a light at infinity
Clear	none	Clear the screen, object lists, and reset the background color.
Trace	none	render the image

Fig. 2. Script Commands: capitalization does not matter. Use ’#’ to comment lines.

The program takes in a specialized input script consisting of the commands detailed in Fig. 2. The script describes an image by detailing the camera, objects, and lights with their positions and physical properties. Each script must begin with a ”Screen” command (possibly following a ”Clear” command) and end with the ”Trace” command. The program will output an image in the TIFF format.

7) Python Libraries: Two Python modules outside of the Python Standard Library were integral to our application. First, kafka-python was needed to connect all our components to the Kafka brokers. It allowed us to create Kafka Producers and Consumers that could send or receive messages. Second, the Pillow module provided us the ability to work with images (still and animated) using Python. It was used to load and save images and also served as the underlying tool to compile frames into a GIF.

B. Stages and Pipeline

Our application can be broken down into 7 stages (shown in Fig. 3):

1) Input Script Development: First, a user must write a series of script files using the commands in Fig 2 (additional details in Section III-A6). Each script will describe a single image frame.

2) Script Transmission: In this stage, the user simply runs our animation_generator.py program with a file name containing the scripts. This will send the scripts to the Kafka service within the cloud, where they will then be pulled out by the Kafka consumers waiting in the ray tracer pods. All these consumers are within the same consumer group, so they essentially work together in parallel to process all of the scripts. There is no more user input after this step.

3) Image Rendering: After a script is processed, the ray tracing program with the same ray tracer pod will run and render the image based on the script.

4) Image Transmission: The final stage within each ray tracer pod is to send the finished image frame back to the Kafka service. The pod will then reset to process another script. Each image frame is then pulled from Kafka by two different GIF compilers; one residing within a cloud container and one residing locally on the user-side. The GIF compilers wait for all frames of the animation to be received, but assume that no longer than 20 seconds will elapse between receiving 2 frames of the same animation. Once that time has been exceeded, they will move onto the next stage.

5) GIF Compilation: In this stage, all image frames are compiled into a single GIF animation. Here, we will detail why we have two GIF compilers in different locations. The reason for our cloud GIF compiler is explained in Section II-B. However we discovered a bug in the Python library we used to render images, Pillow, that would cause corruption in the colors of some frames [7]. We found that this bug was easily avoided when the images were compiled locally, so we decided to implement both for flexibility in future development.

6) GIF Transmission: This stage only applies to the GIF compiled within the cloud. After the compilation is complete, the GIF is sent to the Kafka service once again and pulled out by a GIF consumer waiting on the user-side. The cloud GIF compiler will begin waiting for new frames again.

7) GIF Save: Finally, the GIF animations are saved to the user’s computer. The GIF consumer will save its result as ”gifs/possibly_corrupt_result.gif” while the local compiler will save its result as ”gifs/final_result.gif”.

All source files as well as additional details can be found in Section VI-C.

IV. Evaluation

While we have not done extensive testing, we timed and compared the total execution times of a few sample inputs. Our results are displayed in Fig. 4. We tried two different values for the number of ray tracers and partitions. We also timed how long it takes to render each image frame sequentially in the local computer without even including the GIF compilation step. As seen in the figure, more complex inputs with a higher number of objects (spheres, triangles, boxes, and lights) and frames took longer to process. Furthermore, all cloud compilation took longer than local compilation; this

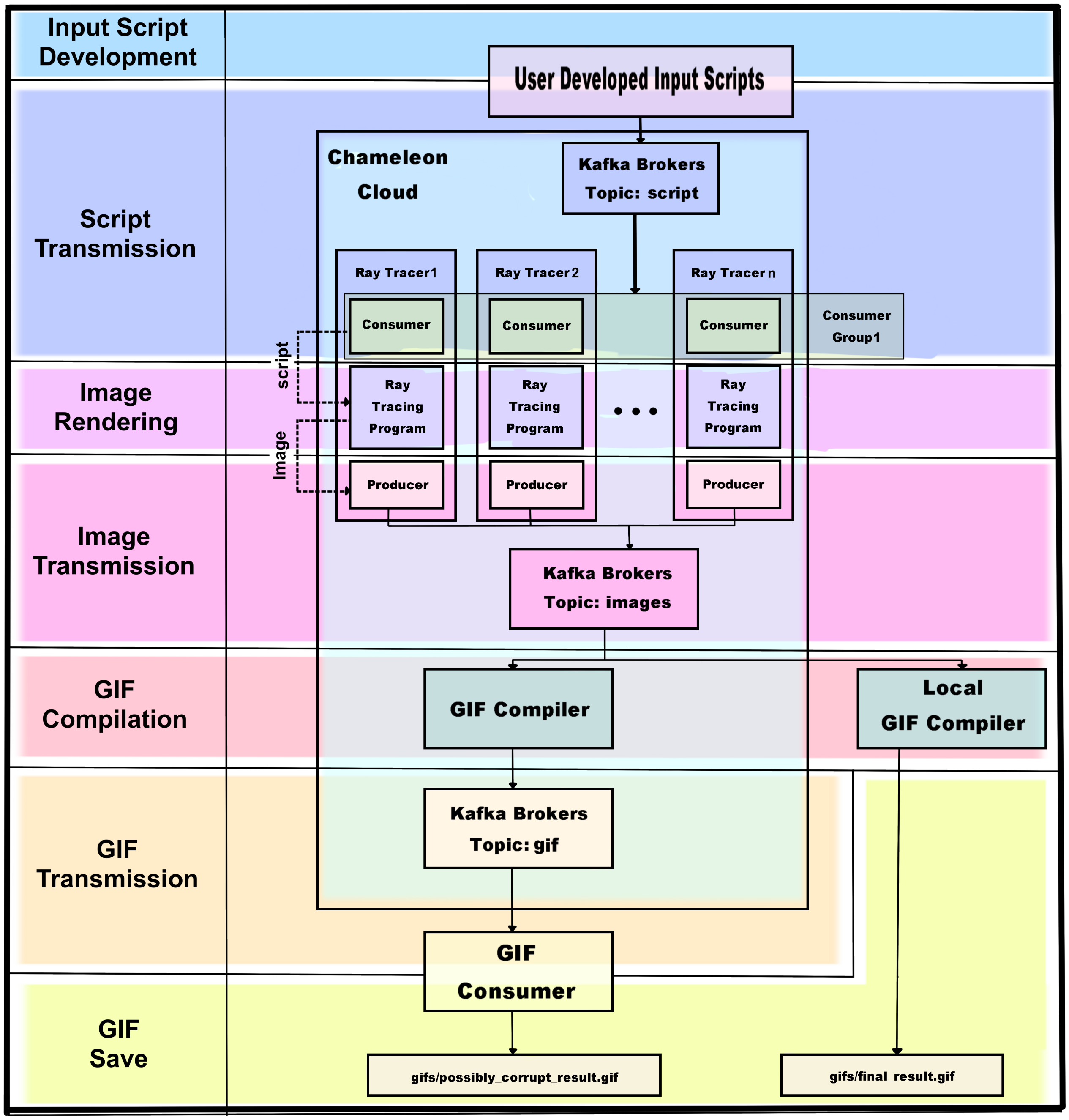
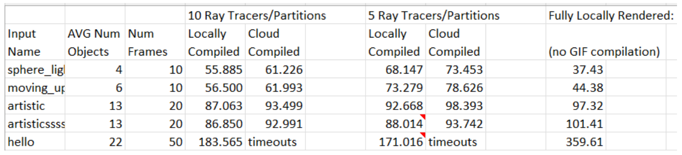


Fig. 3. Visualization of the seven stages of our application along with the full pipeline.



			10 Ray Tracers/Partitions		5 Ray Tracers/Partitions		Fully Locally Rendered:
Input Name	AVG Num Objects	Num Frames	Locally Compiled	Cloud Compiled	Locally Compiled	Cloud Compiled	(no GIF compilation)
sphere_ligl	4	10	55.885	61.226	68.147	73.453	37.43
moving_up	6	10	56.500	61.993	73.279	78.626	44.38
artistic	13	20	87.063	93.499	92.668	98.393	97.32
artisticssss	13	20	86.850	92.991	88.014	93.742	101.41
hello	22	50	183.565	timeouts	171.016	timeouts	359.61

Fig. 4. All of our trial results are displayed here. All times are in seconds. The "AVG Num Objects" refers to the average count of spheres, triangles, boxes, and lights across each script of the input. The locally rendered inputs were not compiled into GIFs, which would increase the execution time marginally. The two results marked with red were likely faster than they should be due to fluctuations in the Chameleon cloud.

is likely because of overhead associated with the additional GIF Transmission stage needed for cloud compilation. As for the different ray tracer and partition numbers, the higher value generally was faster, which is expected due to increased parallelism. Interestingly for the most complex input, the lower number of ray tracers and partitions actually performed better, but it is possible this was due to fluctuations in the Chameleon cloud's processing speeds at the time as this fluctuation was an occurrence we observed throughout the semester on other assignments. Finally, in comparison with the fully locally rendering times, the cloud-based application generally performed better with more complex inputs. This is because with simpler objects, the parallelism of the cloud-based application is overshadowed by the overhead associated with all the message passing.

V. Teamwork

A. Division of Work

When working on the project, we always got together, discussed, and solved each challenge together. Daniel was responsible for most of the actual code writing, Yueqi and Xinyu were responsible for doing research on technologies to be used, helping fix bugs and writing test scripts.

B. Development Platform Used

We created a GitHub repository for code sharing and used Wechat for sharing useful links, ideas, and discuss time to meet. When we were not able to meet in person, we used Zoom to communicate. For writing the final report, we used Overleaf.

C. Testing Strategy

We conducted unit tests by running example scripts or sending example images and GIFs manually for all new components as they were completed. With the current scale of project, we felt confident that manual unit testing is sufficient.

VI. Conclusion

Our cloud-based ray tracing animation generator was able to effectively reduce processing times compared to locally-based rendering. We believe that for more complex inputs, our application would prove to be even more effective.

While we have made considerable progress, here are some future considerations:

A. Limitations

1) Frame Size Limit: The max frame size we currently support is 1024 x 1024.

2) Cloud GIF Compiler Errors: As described in Section III-B5, the cloud GIF compiler experiences abnormal coloring issues due to a known bug in the python Pillow library. We could perhaps use another library for image processing. We have also identified but not yet implemented another workaround.
Furthermore, the cloud GIF compiler takes abnormal amounts of time to respond for larger, complex inputs. We have not investigated the cause yet.

3) Loss of Frames: Occasionally, we observed the loss of 1-3 frames when pulling the rendered frames from Kafka. This was temporarily solved by increasing the consumer timeout of the compilers, but the issue reappeared for inputs of higher frame numbers. We suspect this is occurring during the Script Transmission stage with some scripts not being consumed because there is no consumer assigned to that partition at the time. This may be solved by having backup consumers in the same group, or having another group of consumers altogether. However, the loss of a few frames is generally not noticeable even at our relatively low frame-rate of 10 frames per second, so the extra resources spent on such solutions, especially the latter, may not be worth it. A less costly solution might be to concurrently run the consumer and ray tracing programs within the ray tracer pods so that the consumer will constantly be waiting without stop.

B. Future Work

1) Increase Parallelism of Frame Passing: Currently, sending input scripts to Kafka and pulling rendered frames from Kafka are done sequentially. For inputs of higher frame numbers, both of these processes will generate too much overhead. One solution is to parallelize them by using multiple Kafka producers and consumers.

2) Extend the Ray Tracing Program's Functionalities: Our basic ray tracing program can only render simple shapes include spheres, boxes and triangles. To create a more useful application, we need to extend the functionalities of the ray tracing program, which could include adding more render-able objects, allowing motion blurs, generating softer shadows, and implementing texture mapping.

3) Develop a User Interface: The application currently takes complex input scripts that consist of commands with complex parameters. For users without prior knowledge, it is extremely difficult to understand each command and actually write a script. We would like to build a user-friendly interface so that a wider range of people may use our application. To do so, we may extend out current project to a web-based or mobile application that allows users to design the scene without needing to deal with any numerical parameters and then automatically generates the necessary scripts.

C. Source Repository

https://github.com/danielshu7/animation-generator

REFERENCES

[1] J. Thomas, "What is ray tracing? the games, the graphics cards and everything else you need to know," Available at https://www.techradar.com/news/ray-tracing (2021/12/15).

[2] J. Peddie, *Ray Tracing: A tool for all*. Springer Nature Switzerland AG, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-17490-3

[3] J. Fingas, "Disney explains why its 3d animation looks so realistic," Available at https://www.engadget.com/2015-08-02-disney-explains-hyperion-renderer.html (2021/12/15).

[4] W. D. A. Studios, "Disney's hyperion renderer," Available at https://www.disneyanimation.com/technology/hyperion/ (2021/12/15).

[5] V. Degli-Esposti, F. Fuschini, E. M. Vitucci, and G. Falciasecca, "Speed-up techniques for ray tracing field prediction models," *IEEE Transactions on Antennas and Propagation*, vol. 57, no. 5, pp. 1469–1480, 2009. [Online]. Available: https://doi.org/10.1109/tap.2009.2016696

[6] D. Pohl, "Sponsored feature: Changing the game - experimental cloud-based ray tracing," Available at https://www.gamedeveloper.com/programming/sponsored-feature-changing-the-game---experimental-cloud-based-ray-tracing (2021/12/15).

[7] N. Uchida, "[image-sig] animated gif problem," 2008, bug report. [Online]. Available: https://code.activestate.com/lists/python-image-sig/5245