3D Avatar Generation

Aditya NM[1,2*†], Kavisha Mathur[2,3†], Kevin Jacob[1,2†],
Sri Charan Kanumuri[1,2†], Prafullata Auradkar[1,2†],
Adithya Balasubramanyam[1,2††]

*Corresponding author(s). E-mail(s): adityamuralidhar37@gmail.com;
Contributing authors: kvsha2020@gmail.com; kevin02barca@gmail.com;
royalcharan2002@gmail.com; prafullatak@pes.edu; adithyab@pes.edu;
†These authors contributed equally to this work.

Abstract

With the idea of the Meta-Verse becoming an immediate reality, alongside the development of virtual environments for the Meta-Verse, there is a need to have a unique virtual avatar for each person entering the virtual environment, representing the person behind the avatar. This paper introduces an end-to-end service for accepting a 2D image as an input, a single image, from which a usable virtual avatar with a high-quality model of a person's body and clothes texture, is reliably produced.

The key challenge to such a service is the lack of clarity on what comprises a single execution of the 2D to 3D pipeline, as well as on how to deploy such a pipeline to ensure high throughput. In this paper, the authors provide a complete overview of the various components of the pipeline, discuss the possibilities and challenges of its deployment as a service.t, and produce to a fully rigged 3D avatar that could be deployed in any virtual environment.

Keywords: 3D Model, Virtual Avatar, 3D Waveform processing, Rigging, GPU Containerization, Collaborative Virtual Environments

1 Introduction

The use of 3D avatars has become increasingly popular in recent years, even more so in the entertainment industry. They provide users with more realistic and immersive experiences in video games and movies, and can also be used to create virtual assistants or customer service representatives. In addition, 3D avatars [1] can be used for social media and online communication, allowing users to create a digital representation of

themselves and interact with others in collaborative virtual environments. There is a rapid need for an automated pipeline of creating these avatars for these environments. Further, it should be feasible for this pipeline to be hosted as a dedicated service to generate avatars for users. Such a service could be used by new VR developers to populate their environments. It could even be developed further to provide a uniform interface for end users to enter different virtual environments.

The objective of this paper is to demonstrate an end-to-end, automated pipeline from a simple 2D input to a fully rigged avatar, ready to be used in a virtual environment. The pipeline will take a 2D image of a person as an input, which can then be transformed into a 3D waveform. We then address noisy productions produced by the model by cleaning and smoothing the distorted edges and removing disconnected body parts. The waveform is then imported by Blender, running a Python script with the Blender API for automated rigging. The A pose armature is parented with the mesh of the .obj file. Parenting is the action of associating an armature and its respective mesh. The parenting of the mesh to the waveform results in the formatting of a .fbx file, and is ready to be used in Virtual environments. The avatar is then tested out to work satisfactorily in virtual environments created using the Unity engine. Once deployed as a service, it could be used by new VR developers to populate their environments, and could additionally be used as a part of a uniform interface for users to enter and leave environments, selecting their looks in each one. Remaining sections of the paper are organized as follows- Section 2: Related Work, Section 3: Method, Section 4: Experimental Results, Section 5: Results, Section 6: Algorithms, Section 7: Conclusion, Section 8: Discussion and Future Work and References.

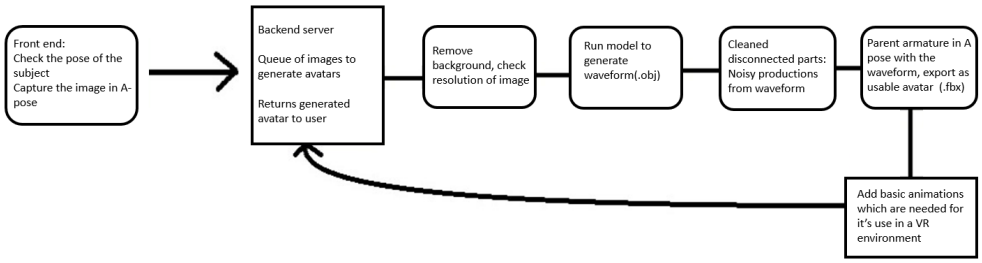

Fig. 1: An overview of the framework

2 Related Work

2.1 Generation of 3D models from 2D images

For general 3D objects, a thread of recent studies train 3D generative models from 2D images, but they mostly work on rendered images with inanimate objects e.g. cars, chairs, etc. In contrast, faces present a distinct challenge due to their rich array of fine-grained details, pose variations, and lighting variations. Several papers have emerged

addressing these challenges specifically for facial image processing. One commonly occurring approach is the utilization of 3D-GAN, which tackles the complexities of facial structures and variations in lighting and poses.

Cutting-edge avatar digitization solutions [1, 2] are based on non-linear models generated from GANs [3] or transformers [4], outperforming traditional linear models which often lack accurate conversion of 2D to 3D [5].

Highly intricate shapes, such as hairstyles and clothing, as well as their variations and deformations, can be digitized in a unified way as cited [6].

Pixel-Aligned Implicit Function (PIFu) [7] is a highly effective deep learning method for implicit representation that locally aligns pixels of 2D images with the global context of their corresponding 3D object. This approach is employed in the pipeline proposed in the current work for the 2D image to 3D waveform stage as it can infer both 3D surface and texture from a single image.

2.2 Automatic rigging

An Armature of a 3D model refers to the system of bones, their motion constraints, and, if any, animation data. This is made manually, from templates, or from scratch. In Blender, a popular way to rig humanoid characters is a plug-in called Rigify. Rigify gives the user an entire bone collection that can be resized and adjusted to work with the user's model. There have been attempts to do this completely automatically for any character. [8] RigNet is an excellent example of a complete automatic rigging system, including appropriate weight painting. It cannot currently handle detailed objects or meshes having vertices of more than 5K. For reference, meshes generated by PifuHD [7] would have a vertex count of over 60K.

2.3 Containerized applications on GPUs

There have been several discussions on optimizing GPU utilization for containers. ConVGPU[9] addresses the problem of multiple containers sharing the same GPU device and implements a scheduling algorithm to prevent deadlocks and ensure that each container is guaranteed the minimum amount of GPU resources it needs for proper execution. GaiaGPU[10] addresses a similar problem of sharing GPUs across different containers in a cloud by partitioning the physical GPU into multiple virtual GPUs, and allocating the V-GPUs based on requests. Taking virtualization to the next step, [11] talks about using the virtualization of GPUs to accelerate GPU-intensive applications in an elastic Kubernetes cluster. The discussions provided in paper regarding GPU utilization are focused on optimizing the architecture of your application to ensure complete utilization of existing GPU resources, without interfering on a kernel level.

3 Method

The methodology is split into two main parts and discussed. The first part involves developing a pipeline to convert a 2D image into a 3D avatar suitable for use in

a virtual environment (Fig. 1). The second part focuses on designing the architecture and methodology to deploy this pipeline on a GPU server, ensuring the highest throughput(Fig. 12).

3.1 Avatar Generation Pipeline from Image Input

The pipeline execution comprises the following distinct stages. The first stage is an image capture stage. The front end of the service captures the image of the user in the few pre-determined poses (such as A-pose, T-pose, etc) it may restrict itself to. Since the mesh generation model subsequently used gives the best results with the subject being in A-pose, input images are restricted to those in A-pose. This is achieved by using MediaPipe and OpenCV in Python. OpenCV (cv2) is used to capture video from a camera, parse frames, and display pictures, and MediaPipe is then used to recognize human postures in real-time. The discovered stances are then analyzed using metrics such as the angle between joints to see if they resemble the A-pose. Until an A-pose is recognized, the associated frame is stored as an image and used as an input image fed to the subsequent stages of the pipeline.

Next, is the waveform generation stage. The image captured is provided as input to the mesh-generation model. We briefly discuss the PiFuHD model [7], an end-to-end multi-level framework that infers 3D geometry of clothed humans at an exceptionally high 1k image resolution in a pixel-aligned manner. Given the 2D image captured in the desired pose, the system needs to additionally recover the backside, which is unobserved in any single image. This calls for the need to predict normal maps for the front and back sides in image space, to be used as additional input. We achieve this by leveraging image-to-image translation networks to produce backside normals, similar to [12–14] There are two main layers in the architecture of the PiFuHD framework: A coarse layer and a fine layer. The coarse layer is instrumental in providing geometrical context for the waveform to be generated and takes images with a lower (512x512) resolution as input and obtains feature embeddings of low resolution (128x128) for the same. The fine layer is responsible for adding more subtle details to the waveform. To achieve this, it takes an image as an input at the original (1024x1024) resolution and encodes it into high-resolution image features (512×512). The second module takes the high-resolution feature embedding and the 3D embeddings from the first module to predict an occupancy probability field.

Once a 3D mesh has been generated, we move on to the Waveform processing stage. In this stage, post-processing steps are implemented to remove noisy productions produced by the model, identified as those disconnected from the main waveform. Leveraging the Blender Python API, the .obj file is scrutinized for active meshes, the primary mesh is identified, any extraneous or disconnected parts are deselected, and the model is rendered editable for fine-tuning coherent portions of the mesh. Metrics such as the area of the model before cleaning, the area of the connected portion, the area of the distorted portion (if any), and the ratio of connected to the original area, are then printed out. The above process is tested out with multiple test cases, including 3D avatars of both genders, various types of clothing, and poses.

Finally, the rigging stage. Here, a pre-designed A-pose armature is parented to the cleaned waveform from the previous stage. An armature is a 3D asset that acts as the

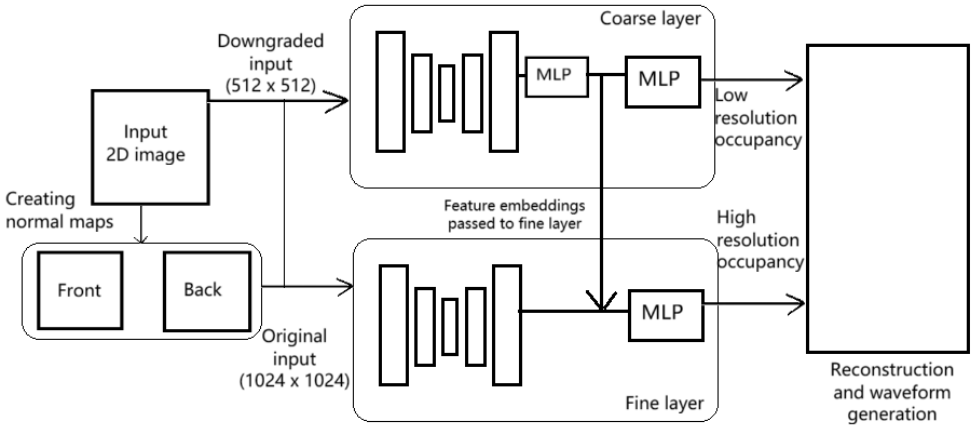

Fig. 2: An overview of the PiFuHD framework

skeleton for a mesh. It does so by warping the mesh of the waveform to move with it. A "weight" in such a relationship is the degree to which the mesh of the child moves with the parent. The higher the weight for a particular area, the less will the mesh move. 3D VR developers typically use a method called "weight painting" where they manually assign weights to these parts to ensure the most natural feel to the moving avatar. We make use of a Blender function to implement this. The function runs an algorithm to estimate the approximate weights required for the particular mesh. A base armature generated by Rigify, a Blender plugin. We modified this armature, by placing it in the A-pose. During testing using various poses, we found that the model generated the cleanest meshes when the input was an A-pose. Once the armature is obtained, animations are added to it in Blender. Initially, an idle swaying animation and a walking animation are made and added to the armature. More animations can be added at any time in the future. The armature and animations are then exported to a .fbx file, which is used to rig the mesh. The mesh is now a 3D Avatar.

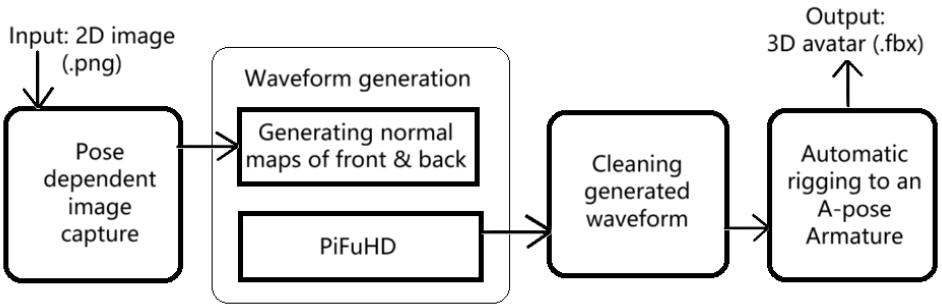

Fig. 3: Overview of stages in proposed pipeline

3.2 Architecture to deploy the pipeline as a service

There are multiple ways for the pipeline to be deployed as a service. A very naive approach would be to have a process listening for requests, triggering the pipeline when a request is made, and then returning the result to the user. The main problem with this approach is the lack of parallelization and the synchronous nature of the application. It will be used as a baseline to accurately evaluate the boost in performance brought about by alternate architectures described in this section.

The pipeline is very compute intensive and makes high demands of the GPU resources of the server. Further, the pipeline in its entirety takes anywhere between 15 to 20 seconds to execute. This immediately rules out a synchronous architecture. When executed on a 16 GB GPU, it was noticed that less than 5 GB of GPU memory was utilized. Thus, such a naive architecture wouldn't ensure the optimum use of GPU resources either.

The architecture proposed is a containerized, asynchronous architecture. The application contains three main types of containers. (1) A flask container that serves as the endpoint for communication between the application and the end users, (2) Several worker containers containing all the binaries and executables required to run a single instance of the pipeline described above, and (3) A redis container which serves as a medium of communication between the different containers, and acts as a backend to store results of different jobs.

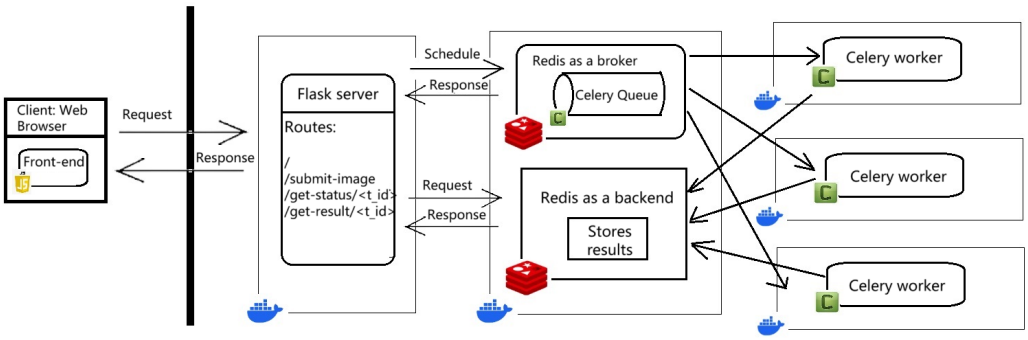

Fig. 4: High-level design of Service

When a request is submitted to the service by the user, it lands on the Flask container. A job ID is created for the request and returned to the user. Updates on the status of the job and the result of the job once it's complete are fetched using the job ID, and displayed to the user. This results in an asynchronous application, ensuring availability to all end-users, and eliminating the possibility of being blocked by a single request. The Flask server uses Celery for distributed workload management, with Redis being used both as a broker, and a backend. Each worker container subscribes to the same celery queue and waits to receive jobs to be executed. To gain control over the parallelization, we specify the argument:

```
CELERYD_OPTS="--concurrency=1"
```

A containerized approach achieves parallelization by distributing different tasks among different workers, allowing their parallel execution and thus, optimizing utilization of server resources. However, it must be done so in a controlled manner. If there are too many parallel executions instantiated, it may lead to a few being starved of GPU resources needed.

It is preferred over other approaches which can achieve parallel execution of jobs, such as multi-processing approaches, because of the ease of horizontal scalability of this approach over multiple instances. Further, containerization also allows an easy way to monitor and limit the number of parallel executions. With the high demands on GPU memory by each process, the performance of each task may degrade considerably if parallelization is left unchecked.

4 Algorithms

The following are algorithms and pseudo-codes for 3D-generated mesh cleaning, Avatar rigging, and the Flask frontend server.

1. **Mesh Cleaning Algorithm**: This algorithm is designed to remove any extra or detached bits of the generated mesh. Ideally, there would be only one mesh and no disconnected parts. In our observation, such disconnected parts, or in other words, fragmentation of the mesh occur when the input image is clear enough, or more likely that the outline i.e., the silhouette of the person in the input image is not clear enough.

Algorithm 1 Mesh Cleaning Algorithm

Require: Active mesh object
Ensure: Cleaned and separated mesh
1: **if** active object is valid mesh **then**
2: Store original area
3: Deselect all objects
4: Select and activate the original object
5: Enter Edit Mode
6: Select all geometry
7: Separate loose parts
8: Exit Edit Mode
9: Calculate cleaned area
10: Remove disconnected objects
11: Deselect all objects
12: **else**
13: Print error message

2. **Auto-Rigging using Blender API**: The algorithm uses Blender as a medium, to parent the armature to the generated mesh of the avatar. Once the parenting is done, the mesh must be given appropriate weights, to ensure it moves as naturally

Algorithm 2 Auto-Rigging using Blender API

1: Create a new Blender file with default objects removed
2: Import the armature from the specified FBX file
3: Select the armature and switch to Pose mode
4: Import the mesh from the specified OBJ file
5: Select the armature
6: Switch to Object mode and select the mesh
7: Parent the armature to the mesh
8: Switch to Object mode
9: Save the Blender file (.blend) to the specified output path
10: Switch to Pose mode
11: Export the scene as an FBX file (.fbx) to the specified output path
12: Switch to Object mode

Algorithm 3 Writing the Flask application.

Require: Flask Installed
1: Define a Celery instance for background task processing
2: Create a route to serve the front-end web page
3: Define route 'submit_image' for receiving image uploads
4: Process the image using Celery by sending a task to the worker
5: Create route '/simple_task_status/task_id' to check task status
6: Create route '/simple_task_result/task_id' to check the result of the task.

Algorithm 4 Celery Worker Algorithm

1: Start Celery worker instance
2: Define a Celery task named 'avatarGen' for generating avatars
3: Receive input image in base64 format
4: Save the input image locally
5: Execute a script (e.g., 'RUN.sh') to generate the avatar
6: Read the output avatar file and encode it in base64 format
7: Return the base64-encoded avatar data as the task result

as possible. Normally, this is a very manual process, but it ensures the results are of the highest quality. We, however, will be automating the process but making use of the "ARMATURE-AUTO" feature on Blender. It assigns weights to each section of the mesh based on various factors, like the shape and size of a section.

article algorithm algorithmic

5 Experimental Results

The proposed pipeline was assessed with a series of experiments that evaluated the quality of the model and its usability in a basic VR environment created by Unity. The pipeline was then deployed in the above-mentioned architecture on a server

of specifications: 16 GB Quadro GPU with 128 GB of additional RAM. Different configurations were analyzed to arrive at an optimum number of parallel containers to be deployed.

5.1 Experimental results of the different stages of the pipeline

The image capture module successfully obtained images in real-time in A-pose that provided the data for the subsequent stages of the pipeline.

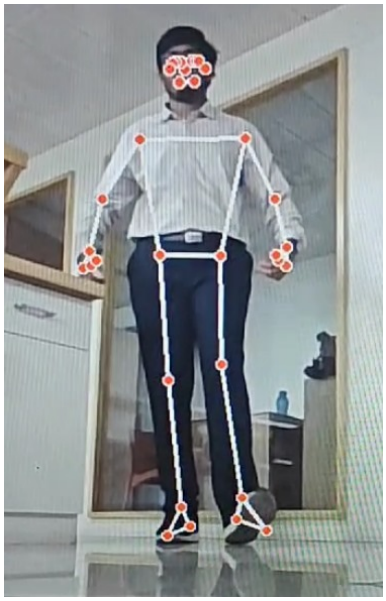

Fig. 5: Image showing how the key points are detected before capturing the image

Before proceeding to the mesh generation model, we first attempt to recover information on the backside of the subject, unobserved from any angle, and generate normal map predictions to further feed as input to the PiFuHD model.[7] The PiFuHD model could capture various anatomical details, for both genders, different textures of clothes, poses, and various lighting. It gave the best results in A-pose, a reflection of the training data that was used while training the model. Large deviations from this pose, such as the T-pose had massive distortions being produced.

The custom cleaning script was tailored to address the issues of disconnected parts in reconstructed models, which could occur due to shadows, other objects in the background, and ambiguous results by PIFuHD. It revealed a significant reduction in artifacts and irregularities, showing the efficacy of the cleaning process.

To ensure the models are well-defined we set a threshold of the ratio between the area the connected to original to be 0.9. This methodology helped us ascertain

Fig. 6: Images, along with the normal maps created for it, which will be fed into the model

that the model produced by PIFuHD has captured the correct aspects from the image provided to it and eased the process of removing disconnected portions.

In the rigging phase, we used a custom-made armature to achieve automatic rigging. After the armature and mesh are parented, the use of Blender's Auto Weight painting functionality creates satisfactory weights for the generated 3D avatars.

The reason, our method of rigging works better at least in our use case is time. The time taken to parent an armature to a generated mesh is a few seconds. But if we were to generate a new armature for every mesh, that would be computationally unfeasible to do so, especially if we were to deploy this pipeline as a service onto the cloud. Thus, by limiting the pose in the input image, we can use a standard armature to rig all the generated armatures.

Finally, the fully rigged avatar is imported into a basic Unity space. The armature used for the design of the pipeline supported two basic movements. One is for in-place movement to mimic realism when the subject is static, and another walking animation to mimic the hand and leg movements when they are walking.

The avatar is imported in Unity, and the animations mentioned earlier on the final avatar which is created by the service, can be satisfactorily viewed.

5.2 Evaluation of architecture used to deploy the pipeline

Table-1 gives an overview of the system configurations of the machine the service was deployed on. A single execution of the pipeline makes a demand of approximately 6 to 7 GB of GPU memory. The utilization of GPU resources is monitored

Fig. 7: Models generated by PIFuHD

by running Python's gpustat as a background process during testing. The data collected by this process is then parsed and desirable graphs are generated using Python's matplotlib.

Different configurations of the architecture are deployed, varying the number of Celery workers with each, and tests are run to monitor the GPU usage and find an optimum setup that completely utilizes the GPU resources. There is a trade-off to be made between the number of containers that we can have running in parallel, and the performance of a single instance of the pipeline. Table-2 provides a summary of the different configrations, as well as the average turnaround times.

The execution of the pipeline on the server using a naive architecture, with a single process listening for requests, and synchronously serving the requests, is taken as a baseline. Any architecture deployed is worth the overhead only if it beats this naive architecture. It is observed that the pipeline would take 17 to 18 seconds to run (the GPU usage during the execution is visualized in Fig.12). Thus, this is a reference benchmark, with which all other carchitecture configurations will be compared with to arrive at an optimum configuration.

After containerization, it is observed that the time increases nearly twofold. Fig.13 shows a comparison between the pipeline's execution on the host, and its execution on a docker container.

It is observed that for a single image being submitted to the service, direct execution on the host would have a better turnaround time, than a containerized approach.

But, when there is an extremely large number of requests being made in one go, the turnaround time increases as a factor of N. The pipeline is evaluated with configurations, varying the number of containers executing tasks in parallel by gradually increasing it from 1 to 4.

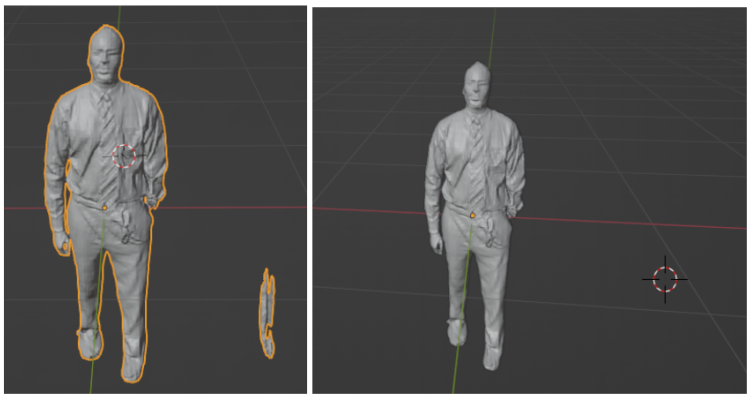

Fig. 8: (a) 3d model with disconnected arm before cleaning (b) 3d model after cleaning no disconnected parts (c) Metric to determine the portion of the cleaned object to the uncleaned object giving a score to the cleaning procedure.

Table 1: Overview of system configurations of machine used to host the service

System component	Capacity
GPU (Quadro) Memory	16 GB
CPU Cores	40
RAM Memory	128 GB
Disk space	1 TB

To better understand the effect of parallelization and to establish an optimum architecture, a bulk request is sent to the service. Python-selenium is used to automate the submission of images to the pipeline through the front end. The performance of the service is evaluated for 15 images, submitted almost instantaneously, one after the other. For the first two cases, where a single container is running, and when two containers are running, the GPU utilization graphs are shown in Figure 12. The average turnaround time, calculated as (the total time to service N requests)/N, is 34 seconds and 20 seconds respectively. Thus, both these architectures fail to beat the baseline naive architecture, simply because the latency added to the execution by containerization is much greater than the benefit of parallelism (in the approach which uses two containers).

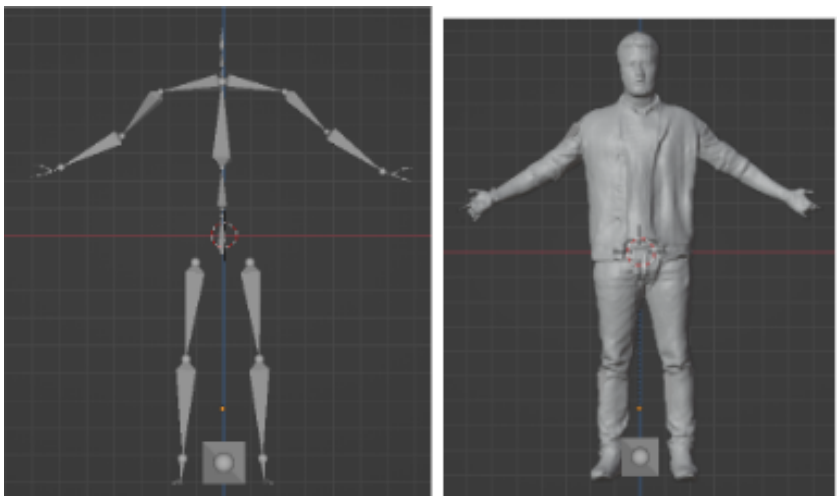

Fig. 9: Manually created skeleton/armature. This

Fig. 10: A generated mesh for an avatar

However, when the architecture that deploys 3 containers and 4 containers respectively are examined, an average turnaround time of 10.6 seconds, and 13.3 seconds respectively are observed. Both approaches beat the baseline, with a speedup increasing with the number of requests being submitted in a single second. However, additionally, an observation is made that the 3-container approach was more optimum in comparison to the 4-container approach. This is largely because the degradation in performance of a single execution of the pipeline (caused due to a lesser GPU resources being allocated to that container) was not worth the speedup achieved through parallelization.

Fig. 11: Model deployed in Unity

Fig. 12: Model animated in Unity

Further, examining the GPU utilization graph for the deployment of 4 containers, we see that nearly half the requests were serviced in the first 70 seconds of execution, while the other half took up the next 130 seconds of execution. This was because while the resources were allocated without any errors initially, the approach eventually reached a breaking point, with 2 of the containers crashing after a CUDA exception error was thrown.

Isolating the last 130 seconds in Figure 12 (d) which describes 4 containers running in parallel, and comparing it with figure 12 (b) which describes 2 containers running in parallel, we find that the two graphs are exactly alike, supporting the fact that when we try to run 4 containers, after a while, 2 of them crash due to being starved of GPU resources and only 2 containers remain operational.

6 Results

An end-to-end pipeline is proposed which takes a 2D input from the user and generates a usable avatar as 3D output. Towards this, PiFuHD is identified as an

optimum mesh generation model, a custom script to parent the mesh with a predefined armature is written, enabled by fixing the position of the subject in A-pose. The automatic rigging approach excels over other alternatives in its simplicity, effectively utilized within the proposed pipeline to construct 3D avatars of human users, with the generated meshes exhibiting near uniformity in terms of rigging considerations.

We also quantitatively show the high demands made on the machine by deploying the pipeline and arriving at an optimum configuration of 3 parallel containers for a single machine with 16 GB of GPU Memory.

For single-machine considerations, a baseline of a naive, non-containerized architecture, is proposed and it is shown that the server must have significant GPU resources for parallelization through containerization to yield better results. This is mainly due to the degradation in performance that comes with fewer resources available, as well as the latency added due to containerization.

For multiple machine cluster considerations, the architecture proposed would be ideal as it is also horizontally scalable.

7 Conclusion

Through this paper, a pipeline is proposed that can create 3D avatars or characters with just a simple input of a picture. Currently, the system produces the best results when the input picture has little to no background noise and well-lit conditions. As on this work, only two animations have been implemented for the avatars: an idle animation to mimic realism with subtle movements when the subject is stationary, and a walking animation.

Baselines and insights on the nature of the demands made by this pipeline on the resources of a single server are also provided. A complete view is thus offered on how one could optimize their architecture to fully utilize the resources on their system, as well as the hardware requirements needed to deploy this as a service on a large scale with a requirement to ensure very high throughput.

8 Discussion and Future Work

Through this paper, we proposed a pipeline that successfully constructed a 3D Avatar along with its rigged model, which is user-friendly.

To further improve on the likeness of the avatar we can emphasize particular features of the body, such as hair [15], face[16, 17]. We could, in the future, also add a text-guided approach to generate 3D shapes or Avatars[18] and texture transform the clothes [19, 20].

The pipeline is limited in terms of the variation in subjects (Only humans in A-pose are supported), and in terms of the variety of animations offered by the pre-designed armature (Only idle pose animations and walking motion animations have been implemented). A video-based motion capture and animation feature can expand the animations supported by the armature as well as make it easier to incorporate new ones based on the requirements of the user, further heightening the end user experience.

While the architecture proposed is horizontally scalable, it's actual deployment on a cluster, along with an accurate measurement of the communication overhead that is added to a cluster deployment of the service is yet to be evaluated.

Finally, while we have discussed how different configurations of the architecture would respond to different workloads (with the naive method outperforming everything when a single request is sent in isolation, but terribly failing short of other methods for bulk requests), the reality would be that of varying workloads with spikes and sudden drops in the number of requests. It would thus, make more sense for the number of workers in the service to be dynamic in a real world setting where the number of requests made to the service vary drastically. An improvement can then be made by using a container orchestration tool like Kubernetes to dynamically bring up or put down containers based on the number of requests which received by the service. With the rise in AI, we can also utilise prompt engineering, to make it easier for users to give a prompt and get results.

Acknowledgements. We would like to express our appreciation to the contributors of the open-source community whose libraries and tools we used for the foundation of the research. This work is dedicated to the broader Virtual Reality Community.

References

[1] Zhang, J., Jiang, Z., Yang, D., Xu, H., Shi, Y., Song, G., Xu, Z., Wang, X., Feng, J.: Avatargen: a 3d generative model for animatable human avatars. In: European Conference on Computer Vision, pp. 668–685 (2022). Springer

[2] Hong, F., Chen, Z., Lan, Y., Pan, L., Liu, Z.: Eva3d: Compositional 3d human generation from 2d image collections. arXiv preprint arXiv:2210.04888 (2022)

[3] Aggarwal, A., Mittal, M., Battineni, G.: Generative adversarial network: An overview of theory and applications. International Journal of Information Management Data Insights **1**(1), 100004 (2021)

[4] Mo, S., Xie, E., Chu, R., Hong, L., Niessner, M., Li, Z.: Dit-3d: Exploring plain diffusion transformers for 3d shape generation. Advances in Neural Information Processing Systems **36** (2024)

[5] Mezghanni, M., Boulkenafed, M., Lieutier, A., Ovsjanikov, M.: Physically-aware generative network for 3d shape modeling. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 9330–9341 (2021)

[6] Hu, T., Yu, T., Zheng, Z., Zhang, H., Liu, Y., Zwicker, M.: Hvtr: Hybrid volumetric-textural rendering for human avatars. In: 2022 International Conference on 3D Vision (3DV), pp. 197–208 (2022). IEEE

[7] Saito, S., Simon, T., Saragih, J., Joo, H.: Pifuhd: Multi-level pixel-aligned implicit function for high-resolution 3d human digitization. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 84–93 (2020)

[8] Xu, Z., Zhou, Y., Kalogerakis, E., Landreth, C., Singh, K.: RigNet: Neural Rigging for Articulated Characters (2020)

[9] Kang, D., Jun, T.J., Kim, D., Kim, J., Kim, D.: Convgpu: Gpu management middleware in container based virtualized environment. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 301–309 (2017). IEEE

[10] Gu, J., Song, S., Li, Y., Luo, H.: Gaiagpu: Sharing gpus in container clouds. In: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pp. 469–476 (2018). IEEE

[11] Naranjo, D.M., Risco, S., Alfonso, C., Pérez, A., Blanquer, I., Moltó, G.: Accelerated serverless computing based on gpu virtualization. Journal of Parallel and Distributed Computing **139**, 32–42 (2020)

[12] Gabeur, V., Franco, J.-S., Martin, X., Schmid, C., Rogez, G.: Moulding humans: Non-parametric 3d human shape estimation from single images. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) (2019)

[13] Natsume, R., Saito, S., Huang, Z., Chen, W., Ma, C., Li, H., Morishima, S.: Siclope: Silhouette-based clothed people. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2019)

[14] Smith, D., Loper, M., Hu, X., Mavroidis, P., Romero, J.: Facsimile: Fast and accurate scans from an image in less than a second. In: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) (2019)

[15] Zhang, M., Zheng, Y.: Hair-gan: Recovering 3d hair structure from a single image using generative adversarial networks. Visual Informatics **3**(2), 102–112 (2019)

[16] Luo, H., Nagano, K., Kung, H.-W., Xu, Q., Wang, Z., Wei, L., Hu, L., Li, H.: Normalized avatar synthesis using stylegan and perceptual refinement. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 11662–11672 (2021)

[17] Shi, Y., Aggarwal, D., Jain, A.K.: Lifting 2d stylegan for 3d-aware face generation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and

Pattern Recognition, pp. 6258–6266 (2021)

[18] Cao, Y., Cao, Y.-P., Han, K., Shan, Y., Wong, K.-Y.K.: Dreamavatar: Text-and-shape guided 3d human avatar generation via diffusion models. arXiv preprint arXiv:2304.00916 (2023)

[19] Mir, A., Alldieck, T., Pons-Moll, G.: Learning to transfer texture from clothing images to 3d humans. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 7023–7034 (2020)

[20] Yin, K., Gao, J., Shugrina, M., Khamis, S., Fidler, S.: 3dstylenet: Creating 3d shapes with geometric and texture style variations. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 12456–12465 (2021)