Efficient Photorealistic Avatars using ML/AI

Cyrine Boudaya
Technische Universität Berlin
Berlin, Germany
cyrine.boudaya@campus.tu-berlin.de

Belinda Myteberi
Technische Universität Berlin
Berlin, Germany
belinda.myteberi@campus.tu-berlin.de

Rebecca Charlotte Barth
Technische Universität Berlin
Berlin, Germany
Rebecca.Barth.28@gmail.com

Minh Khoa Doan
Technische Universität Berlin
Berlin, Germany
minh.k.doan@campus.tu-berlin.de

***Abstract*—Within this project, we created prototype that constructs 3D avatars of a person and applies gestures to it. The prototype was created from a single video input. Through our work we aimed to use neural networks, which ensure the detection and the transfer of facial expressions from the person to a 3D avatar. During this project we explored two different approaches to create a dynamic 3D avatar. One of them being Hash-NERF, making use of a neural radiance field to create an implicit 3D scene representation and Neural-Head-Avatars, based on flame mesh technology. Our final solution is based on neural-head-avatars with an integrated Application on top for an optimised user experience.**

***Index Terms*—avatar, machine learning, human, video , facial expressions, interface**

I. Introduction

Human animated avatars with expressive movements are today highly demanded as virtual characters in telecommunications and in fields of VR/AR animation , entertainment, healthcare sector, human computer interaction and authentification.
In fact the expressive animation of avatars has benefited from the advances of data-driven deep learning and motion capture technologies. Although this can be considered as a sophisticated solution, one important advantage for mobile applications is that it significantly requires lower Bandwidth . Moreover a 3D avatar can actually be easily used and integrated in several fields such as in games or virtual meeting rooms, with changing geometry, lighting or viewpoint [1].
Thus considering its important role , covering all the details that would be specific and differ from one user to an other such as hair, clothing, facial features and shape should be the main aspects of the provided solutions to the end users [2].
In our proposed method, we are tackling this challenging task based on a short portrait video sequence of a person which will reconstruct the model with its explicit geometry.

II. Hash-Nerf

Based on previous work on encoding, the hash-Nerf method aims to eliminate their drawbacks and to overcome their limitations. It starts by storing the trainable feature vectors in a compact spatial hash table. The size of the hash table is a hyper-parameter *T*, which can be set so that number of reconstruction quality parameters can be adjusted. The outputs which are generated by the indexed hash Tables will be combined and concatenated and then trained within MLP (Multilayer perceptron). [7]
Furthermore the network handles the hash collisions for the 3D Reconstruction in order to enhance the performance and to prevent the control flow divergence.
The features of the neural network of this approach are :

- weight parameters Φ
- encoding parameter θ
- L levels containing T feature vectors with dimensionality F. [7]

A spatial hash function will be used :

$$\text{h(x)} = \Big(\bigoplus_{i=1}^{d} x_i \pi_i\Big) \bmod T$$

with $\oplus$ representing the XOR operator , π_i the large prime numbers , d the different dimensions of the input vector and T the maximum number of entries pro level in the hash table.
Moving now to the architecture of the network . We shall see that Multilayer Perceptron is the basic unit of the Nerf model. The model involves a **density** MLP, which consists of one hidden layer and maps the hash encoded position y = enc (x, Φ) and then generates 16 output values. Moreover the second part consists of a **color** MLP, which is based on two hidden layers, takes the output of the first component (density map) besides to the view direction, which is projected into 16 coefficients to provide a view-dependent color variation. [7]
For the remaining functions , we see that the MLP, composed of two hidden layers , uses the rectified linear unit (ReLU) as an activation functions on the hidden layers.
Once computed the result of the encoding performed by model for each resolution will be then concatenated.
One aspect, which is important to mention is that the choice of the hash table size should be well considered and not arbitrary. This is due to the fact that *T* has a considerable effect not only on the memory but also on the quality and the performance of the model. [7]
We justify the choice of the model because it proved a better

and quicker performance compared to prior work. As an example if we take ACORN (Adaptive Coordinate Networks for Neural Scene Representation) achieved a PSNR (single de Peak Signal to Noise Ratio) of 38.59 db after 36.9h. At the same time, the hash-Nerf realized the same result after 2.5 minutes based on te same input parameter (T = 2^{24}) [7]

III. Monocular video input

Since the main assumption of the Nerf is that the scenes are not dynamic, we decided to integrate other submodules to ensure the dynamicity of the input scenes. Our code is based on a PyTorch port of the official Tensorflow NeRF code. The camera parameters will be estimated with the help of the Structure-from-Motion (SfM) implementation of COLMAP once not given. [8]
COLMAP is a general-purpose Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline providing a graphical and a command-line interface. [9]
The main idea of the model is that it , based on RGB images of a dynamic scene represents them in a a high-quality space-time geometry and appearance.
Non-rigid scenes will be represented by two main elements: (1) a canonical neural radiance field for capturing geometry and appearance and (2) the scene deformation field.
The architecture of the network is analogous to the Nerf architecture. Starting with the **Ray-Bending** Network,it is based on a 5-layer MLP containing 64 hidden dimensions. For the second component , the **rigidity** network it is 3-layer MLP with 32 hidden dimensions. For both networks, ReLU is used as an activation and all the weights for the last layer will be initialized to zero. The last layer's output is ultimately passed through *tanh* activation function and then shifted and reranged so that it is located between zero and one . [8]
To evaluate the performance of the model, the authors of [8] compared ot to the rigid Nerf . While the rigid Nerf showed some blur by the input reconstruction task, the Non-rigid Nerf achieved qualitatively better results by the representation. It still shows some artifacts , which are unwanted but still better than the Rigid-Nerf.
Quantitavely the non-rigid Nerf scored the best values for SSIM (Structural Similarity) and LPIPS (Learned Perceptual Image Patch Similarity) and the second-best PSNR after the naïve NR-NeRF.
Although it has proved good results quantitatively and qualitatively , it is still considered as a slow solution. Besides another limitation , which is inherited from the Nerf Model is that the background and the foreground are required to be close to each other. [8]

IV. Neural-head-Avatars

Neural Head Avatar is an explicit approach to model a human avatar. We aim to create the avatar from a monocular RGB portrait video made by a webcam or a basic mobile phone without a multi-view camera setup. The output of our solution can be integrated into various graphics pipelines and the only prerequisite is that the pipeline must use triangular meshes.

A. FLAME MESH

This approach uses the FLAME Mesh head model [10] as a geometric backbone but with some adjustments, such as removing the lower neck, adding faces to close the mouth cavity area, and uniformly four-way subdividing the face. These adjustments increase the number of vertices that are used to create the triangle mesh from 5023 to 16227. [5]
The input video is split into consecutive frames and each frame generates a shape β , pose ϕ and expression ψ parameters. We utilize these parameters to tune our FLAME MESH model.

$$V_{flame} : R^{300}, R^{100}, R^{3k} -> R^{16227*3}$$
$$\beta, \psi, \phi -> V_{flame}(\beta, \psi, \phi)$$

B. Two Feed-forward MLP Networks

1) Geometry Refinement Network G : FLAME model [10] creates a mesh excluding hair and facial details but these two components are very important in creating a photorealistic human avatar. To tackle the hair and facial details problem a **Geometry Refinement** Network is implemented. As inputs for the network are used the normalized 3D coordinates of the vertices on the template mesh. The pose ϕ parameter is added to G to ensure the parsing of dynamic geometry effects and then G(ϕ) is added to the whole Vertices V of the mesh.

$$G : R^{3k} -> R^{152273*3}$$
$$\phi -> G(\phi)$$
$$V_{(\beta,\psi,\phi)} = V_{flame(\beta,\psi,\phi)} + G(\phi)$$

2) Texture Network T : **Texture** Network T enables the synthesis of view- and expression-dependent effects by predicting the color of a point in the geometric mesh. The color of the point is received by feeding into a SIREN MLP the 3D coordinates of the FLAME and a surface embedding vector which is sampled from a discrete 2D grid in UV Space. Features that were extracted by the Flame expression and pose parameters are then fed into a mapping network in order to predict the frequencies and the phase shifts of the sinusoidal activation function in the FiLM layers of the SIREN MLP in order to achieve synthesizing of a dynamic texture. To get the predictions of RGB within the range of -1 and 1, the output is passed into a *tanh* activation function. So in conclusion, it needs as input the 3D coordinates of the point in the mesh, the pose expression parameter of the particular frame, and a local patch of the rendered normals. The output of the network is an estimated color value that conditions the model for every specific view- and expression-dependent effect. [5]

$$T : R^3, R^{100}, R^{3k}, R^{n*n*3} -> R^3$$
$$p_i, \psi\, \phi, \hat{N}_i -> c_i$$

p_i represents a 3D location on the canonical FLAME mesh depicted in i, $\hat{N}_i$ is a local patch of the rendered normal map around i , c_i shows the predicted color at pixel i.

C. *Optimization of the Input Data*

The joint optimization of head geometry and texture is an underconstrained optimization problem meaning that there are several possible solutions that can satisfy the given constraints. Short monocular video sequences provide limited information for reconstructing the 3D model. Therefore, it is difficult to obtain an accurate model from these sequences. The reconstruction of texture and color is also challenging, as the appearance of the object may vary in different frames of the video sequence. To overcome this problem, regularization strategies are used to achieve the smoothness of reconstructed surfaces and view-consistent texture synthesis. [5] The joint optimization function (E_{joint}) is defined as the sum of the geometry error (E_{geom}) and appearance error (E_{app}).

$$E_{joint} = E_{geom} + E_{app}$$

E_{geom} calculates the data and regularization terms for the geometry like measuring the distance between 2D facial landmarks [11] and their projected counterparts on the mesh, comparing the image-space Laplacians of predicted and pseudo-normal maps, matching the semantic regions of the input and reconstructed mesh for facial skin, ears, neck, eyes, hair, and regularize the FLAME parameters and the geometry MLP G using statistical properties of the linear shape model and a Laplacian regularizer.

On the other hand, E_{app} specifies the terms for texture and color reproduction. It represents how well a neural head model can reproduce the color and texture of a human head image, and how visually similar the generated image is to the ground truth image. E_{app} consists of energy terms of photometric consistency which are achieved by comparing the pixel values of both images, an energy term for perceptual distance, and weights to control the balance between photometric and perceptual consistency in the generated images.

The objective of joint optimization is to minimize both geometry and appearance errors simultaneously to achieve a photorealistic avatar as a final result.

V. Performance and Challenges

Optimizing one avatar in a reasonable amount of time requires CUDA acceleration. The original authors of the project detail a 6 hour optimization process based on their workstation hardware (2x NVIDIA A100) and a dataset composed of 1500 image frames [5]. The same process using a NVIDIA RTX 2070 Super, a consumer-grade and older generation GPU, lasted 2 hours with a significantly smaller sample size of 105. The biggest differentiator being the available video memory, limiting the batch sizes, while training the model. Apart from this step in the pipeline, all other steps were comparatively fast, with the video pre-processing, defining of facial landmarks, and head tracking taking a total of 17 minutes. This discrepancy in performance due to hardware requirements presented a challenge for testing our application. The majority were not able to finish a full run, failing at the head tracking or avatar optimization step (Table I). For some GPUs like intel iRIS Xe, the docker image building failed due to timeout delation while reading a response from the server for the installation of the model library requirements. Even with our one dedicated GPU the latter had to be restarted from a checkpoint and only barely completed using PyTorch's max_split_size_mb parameter, which helps with borderline cases not running out of memory at the cost of performance [12]. Furthermore, some package sources from the original project were no longer available, most notably PyTorch and pytorch3d. Having to collect these packages manually, resulted in a lengthy process analyzing which packages were compatible with each other, as well as the existing code base of Neural Head Avatars. Adding in the differences in operating systems between the team members, this hindered the progress towards creating a unified code base.

TABLE I
Performance Details (Time per step)

Hardware	**Steps**			
	Landmarks	***Head Tracking***	***Optimization***	***Reenact***
2070 Super	00:01:01	00:14:32	01:56:00[1]	00:02:03
intel 6 core cpu (without NVIDIA)	00:05:00	-	-	00:03:00
intel I9 Gen 10, Nvidia M5000	00:04:00[2]	not yet finished	not yet finished	not yet finished

[1]combined time after restart from checkpoint
[2]computed for a 6 seconds video

VI. Final Solution and Improvements

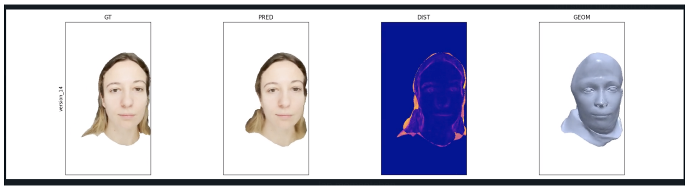


Fig. 1. Final Solution. The right most column visualizes the Hausdorff distance from our predicted face mesh to the recorded GT.

Our first approach consisted of combining hash-nerf with the nonrigid video pre-proccessing to use hash-Nerf with monocular video input. This was successfully for a non dynamic scene representation but we were meeting several limitations in terms of making the representation implicit and manipulation it it with transformations to achieve a dynamic 3D Avatar. Due to time constraints we needed to pivot towards a different solution using neural heads. The final solution presented in this project involves the integration of a novel neural representation: Neural Head Avatars. As mentioned above neural-head-avatars models consist of three steps:

A. *Pre-processing the monocular video input*

The first step of the pipeline is to record and prepossess a monocular video input. In order to ensure the accuracy of

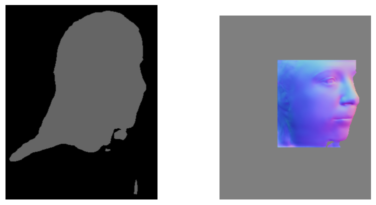


Fig. 2. Segmentation and face-normals of one frame

the avatar's features and movements, it is important to use a high-quality camera with a high resolution and frame rate. This will help capture fine details and movements of the face that are necessary for creating a realistic avatar. Although the background removal is promising great success (See figure 1) the background should be relatively noiseless and the hair should be tied back, to achieve a better result. One has to make sure that only one subject is visible in every frame and that the head is turned as well in both directions up to profile views in order to provide enough information for the avatar optimization [3].

Once the video has been split into frames which is done by a cv2 library, it is necessary to perform some further pre-processing steps. Features annotation is quite crucial for Optimizing the explicit Neural Head Representation later. In order to annotate iris and facial landmarks using a package for face-detection by Google®'s MediaPipe. Those landmarks are used later on for the optimization of head geometry and texture. Furthermore the prepossessing detects corresponding predicted semantic labels and predicted normal maps, which are also creating the base for the next flame mesh and texture optimization.

B. Optimizing the explicit Neural Head Representation

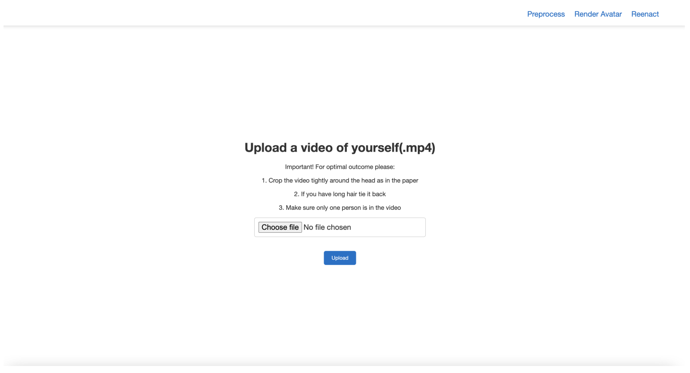
After pre-processing the data for each frame as well as transformation parameters are stored in an output file. As mentioned above the neural head methodology starts off with one generic base flame mesh. The RGB input frame is passed through a real-time face tracker whose purpose is to estimate the basic shape, expression, and pose parameters of the default flame mesh [4]. Through the geometry refinement, **Network G** the model gets iterative adapted. After that using a stand art computer graphics pipeline the final mesh is rasterized with a standard computer graphics pipeline. The mesh is photo-realistic textured through the texture **network T**. In order to achieve that T is conditioned with the canonical surface position, a local normal patch, and flame parameters. Through that T enables the synthesis of view- and expression-dependent effects. However, it does not create a view and expression-independent texture that can be mapped to the generated 3D mesh. The networks and FLAME parameters are optimized by iteratively adjusting the FLAME parameters to fit the input image or video frames with color-dependent and color-independent energy terms. This separates the shape and texture information in the input data, which makes it possible to manipulate them independently. Based on several viewing directions of the personalized flame mesh is rendering a frame in every viewing direction and is textured based on this view. This creates a 3D video as can be seen in Figure one and we additionally exported the created triangle flame mesh in order to make it possible for users to download it an import it to their preferred game engine. [5]

C. Applying facial expressions

For the reconstruction of the avatar and modeling facial parameters, we implemented a script that made it possible to manipulate facial expression and view pose, as only reenactment implementations (of meshes trained on two different people) were available. The reconstructed Neural Head Avatar can be animated using the expression and pose parameters Ψ, Θ, and can be rendered under novel viewpoints. Using a multi-layer perceptron to predict the 3D meshes and dynamic textures, depending on the facial expression and pose of real humans. It is possible to input 4 different facial expression parameters each manipulation of the final flame mesh in a particular way to imitate the expression. This is possible because Neural-Heads output is explicitly represented geometric which allows the synthesis of new expressions and poses for an optimized avatar. This was difficult to implement given the implicit scene representation of NERF or other view synthesis algorithms. [5]

D. Flask

We additionally embedded the neural-head solution in an application to make it user-friendly, and overcome challenges with bad documentation and tweaking configuration files. In order to implement the application, we were using Flask. Flask is classified as a micro-framework because it does not require particular tools or libraries and was therefore useful to integrate it with the python based machine learning model [6]. Using simple html templates and app routing through flask enables the user to navigate through the three different steps of our solution. On the pre-processing tab, the user receives instructions on how to record the monocular video input for the best possible outcome. The user is able to upload the personal mp4 and pre-process the video. Once preprocessed the user has the option to download extracted frames including the important face detection parameters that provide the input for the avatar rendering. (See figure 3)



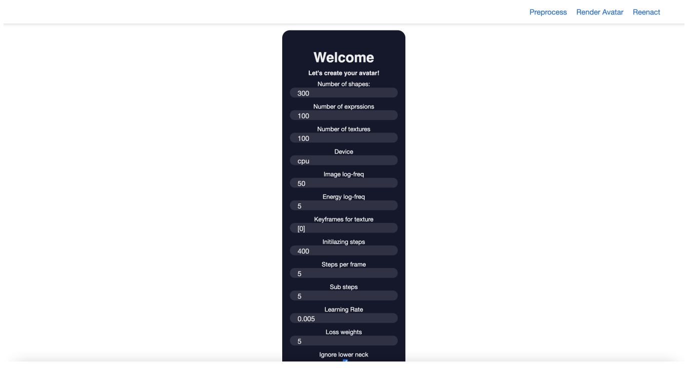
Upload a video of yourself(.mp4)

Important! For optimal outcome please:

1. Crop the video tightly around the head as in the paper
2. If you have long hair tie it back
3. Make sure only one person is in the video

Fig. 3. Upload video

The user can then proceed to the next tab (the preprocessed data is stored locally for now but would be stored in a database once deployed) and can render the avatar. The user based on the knowledge of computer vision algorithms can choose configuration parameters such as the keyframes for texturing, number shapes, iteration, etc. Once the avatar is rendered the user can download the final data sets, the 3D mesh, and the final rendered video (See figure 4).



Welcome

Let's create your avatar!

Number of shapes: 300

Number of exprssions 100

Number of textures 100

Device cpu

Image log-freq 50

Energy log-freq 5

Keyframes for texture [0]

Initilazing steps 400

Steps per frame 5

Sub steps 5

Learning Rate 0.005

Loss weights 5

Ignore lower neck

Fig. 4. Render Avatar input params

Based on the created ckpt and tracking files the user can then proceed to reenact the avatar. The user can choose between four expression parameters: Smile, provocative, clueless, or indifferent. Thereby the user can mix different parameters and also adjust the global view pose as well as the neck pose. The output unfortunately is suffering at times in quality (See figure 5).

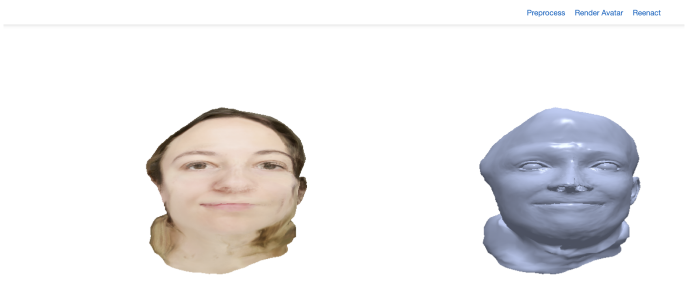


Fig. 5. Smiling expression

E. Docker

As mentioned above we were facing several challenges regarding different operating systems and computational power of our local machines. In order to overcome those and ensure seamless deployment, we dockerized the application, allowing it to run in a Linux environment avoiding Python versioning problems. We opted for a GPU enabled PyTorch base image using PyTorch 1.11.0 with CUDA 11.3 support enabled for an automatic NVIDIA driver and CUDA Toolkit setup. However, this base image locked us into using their provided Python 3.8 installation, which does not meet the required minimum version of 3.9 dictated by the Neural Head Avatars project. We could not determine any loss in functionality, but without matching hardware any possible performance differences could not be investigated. Having this container setup, we can easily run computational heavy operations in the background and our application in the front-end. This also enables us to easily scale the application as needed, making it suitable for use in large-scale applications. Unfortunately, the Docker image needs to be built individually by any potential user due to the licensing of the FLAME models used as a base.

F. Potential improvements

While the final solution is highly effective, there is always room for improvements. One potential area for improvements is the accuracy and realism of the neural representation. Another important improvement could be to optimize the performance of the model in order to achieve the goal to render the meshs texture in real-time based on the video position, rotation and facial expression based on emotions. To enable this one could optimize the neural network architecture and parallelize algorithms to reduce the computational resources required to achieve this goal. Unfortunately all team members did not have access to the write computational ressources therefore a constant more efficient hardware such as graphics processing units could be used to accelerate the rendering process.

The integration of avatars into existing augmented and virtual reality (AR/VR) technologies would be the next step after optimizing the neural network and enhancing the model's realism. This could mean developing applications based on WebGL or OpenGL that let users interact with the avatars in real-time. The avatars could be used to create social networking and gaming platforms as well as more immersive virtual environments like the metaverse for example as we already mentioned.

Another goal could be to deploy our application solution at scale. Therefore we would need to move the application to a deploy-ready state and have the financial means to run the computationally heavy models in a cluster. This would involve optimizing the code, database, and server infrastructure to ensure that the application can handle large volumes of traffic, and user requests and keeping data security in mind.

If we would want to provide the users with a more personalized UX/UI, the implementation of a database is crucial. This would enable the user to save their avatars and data across different sessions and platforms, making it easier to use the avatars across different applications and environments in form of plugins.

VII. Discussion and Outlook

During this project, we got to explore two methods that create a 3D scene presentation based on 2D inputs. Whereas the Hash-Nerf technology has impressive results regarding static scene representations it is difficult to further manipulate the implicit representation. Using flame models as a base for a photorealistic avatar offers the advantage of customisability and real-time manipulation. Most of the state-of-the-art methods to create a photo-realistic scene several are avoiding explicit geometry reconstruction and rely on image or feature-based warping for motion control and generative networks for image synthesis [5]. This creates a vacuum for technologies that focus on manipulatable photorealistic avatars. One learning outcome of this project is the trade-off between the high precision quality of an implicit scene representation and with transformability of explicit representation. The neural-head technologies come close to a solution that uses neural networks to predict texture and adjust the geometry iteratively. Future work and projects could focus on using this technology to integrate further improvements mentioned above such as real-time rendering of the texture in game engines and improving the performance of the algorithms.

REFERENCES

[1] Alexandru Eugen Ichim, Sofien Bouaziz, M. Pauly " Dynamic 3D avatar creation from hand-held video input", 2015

[2] T. Alldieck, M. Magnor , W. Xu, C. Theobalt and G.Pons-Moll "Detailed Human Avatars from Monocular Video", 2018.

[3] Philip-William Grassal*, Malte Prinzler*, Titus Leistner, Carsten Rother, Matthias Nießner, Justus Thies, "GitHub - philgras/neural-head-avatars: Official PyTorch implementation of "Neural Head Avatars from Monocular RGB Videos", 2022.

[4] Tianye Li, Timo Bolkart, Michael. J. Black, Hao Li, Javier Romero, "Learning a model of facial shape and expression from 4D scans. ACM Transactions on Graphics" 2017, (Proc. SIGGRAPH Asia),

[5] Philip-William Grassal*, Malte Prinzler*, Titus Leistner, Carsten Rother, Matthias Nießner, Justus Thies, "Neural Head Avatars from Monocular RGB Videos" 2022

[6] Github, "GitHub - pallets/flask: The Python micro framework for building web applications", 2023

[7] Thomas Müller, Alex Evans, Christoph Schied, Alexander Keller: "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding", 2022

[8] Edgar Tretschk, Ayush Tewari, Vladislav Golyanik, Michael Zollh Öfer, Christoph Lassner and Christian Theobalt "Non-Rigid Neural Radiance Fields: Reconstruction and Novel View Synthesis of a Dynamic Scene From Monocular Video ",2021

[9] COLMAP ,"https://colmap.github.io/"

[10] Li, Tianye and Bolkart, Timo and Black, Michael. J. and Li, Hao and Romero, Javier. Learning a model of facial shape and expression from 4D scans. ACM Transactions on Graphics, (Proc. SIGGRAPH Asia), 36(6):194:1-194:17,2017. https://doi.org/10.1145/3130800.3130813 https://flame.is.tue.mpg.de/" 2017

[11] Adrian Bulat, Georgios Tzimiropoulos. How far are we from solving the 2D & 3D Face Alignment problem? (and a dataset of 230,000 3D facial landmarks) In International Conference on Computer Vision, 2017.

[12] CUDA Semantics, "https://pytorch.org/docs/1.11/notes/cuda.html"