MASTER’S THESIS

**Real-Time Neural Avatar Rendering on Mobile Devices** Master’s Educational Program: Information Science and Technology

Student

Alexey S. Larionov

Information Science and Technology

June 3, 2022

Research Advisor:

Victor S. Lempitsky

Associate Professor

Co-Advisor:

Renat M. Bashirov

Leading Research Engineer

Moscow 2022

All rights reserved.©

The author hereby grants to Skoltech permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

**Нейронный рендеринг аватаров в реальном времени на мобильных устройствах**

Магистерская образовательная программа: Информационные науки и технологии

Студент

Алексей С. Ларионов

Информационные науки и технологии

3 Июня, 2022

Научный руководитель:

Виктор С. Лемпицкий

доцент

Со-руководитель:

Ренат М. Баширов

ведущий инженер-исследователь

Москва 2022

Все права защищены.©

Автор настоящим дает Сколковскому институту науки и технологий разрешение на воспроизводство и свободное распространение бумажных и электронных копий настоящей диссертации в целом или частично на любом ныне существующем или созданном в будущем носителе.

**Real-Time Neural Avatar Rendering on Mobile Devices**

Alexey S. Larionov

Submitted to the Skolkovo Institute of Science and Technology

on June 3, 2022

**Abstract**

During development of artificial intelligence models, the utmost attention is brought to qual ity (accuracy) of their outputs. However, some models are meant to be executed on limited user hardware. Thus, computational resources also require careful consideration, when it is necessary to sustain high performance.

This master thesis describes implementation of an application with an integrated Deep Neural Network, for mobile devices on Qualcomm Snapdragon platform. The neural network generates images of full-body 3D human avatars. The generation takes into account position and orientation of the mobile device. When avatar images are visualized on top of camera frames, the Augmented Reality (AR) effect emerges. A user can observe the avatar from different angles, either from afar or close up. Thus, the generated images should keep the visual quality from angles, unseen in the training data. The first contribution of this thesis is organization of a computational pipeline to achieve real-time performance. Secondly, we carry out research on ways of increasing the visual quality, that would be robust to arbitrary view points.

As a result, we execute the baseline model [36] in the mobile application in real time, with more than 30 frames per second and up to 640 *×* 640 pixels resolution. We generate the model’s input, such that the observed portion of the body occupies as much frame space as possible, thus increasing visual quality in the AR application. We extend the training procedure of the neural net work training, to learn and produce images in either full-body scale or close-up scale. We propose solutions to partially prevent visual artifacts that appear with the baseline training pipeline.

**Keywords:** Real-Time Performance, Mobile Hardware, Neural Rendering, Deep Neural Network, Human Avatar Reconstruction, Computer Graphics.

Research Advisor:

Name: Victor S. Lempitsky

Degree: PhD

Title: Associate Professor

Co-Advisor:

Name: Renat M. Bashirov

Degree: MSc

Title: Leading Research Engineer

3

**Нейронный рендеринг аватаров в реальном времени на мобильных устройствах**

Алексей С. Ларионов

Представлено в Сколковский институт науки и технологий

3 Июня, 2022

**Реферат**

При разработке моделей искусственного интеллекта, наибольшее внимание уделяется качеству (точности) результатов. Однако, некоторые модели предполагается выполнять на ограниченном пользовательском аппаратном обеспечении. Для поддержания их высокой производительности, необходимо учитывать затрачиваемые вычислительные ресурсы.

В данной магистерской диссертации описывается реализация прикладного приложения со встроенной глубинной нейронной сетью, для мобильных устройств на платформе Qualcomm Snapdragon. Приложение визуализирует изображения 3D аватара человека, генерируемые нейронной сетью. Генерация изображений происходит в реальном времени, более 30 кадров в секунду. Изображения аватаров учитывают пространственное положение устройства, и при визуализации поверх изображений с камеры, образуется эффект дополненной реальности (Augmented Reality, AR). Пользователь может осматривать аватар с разных углов, в отдалении и приближении. Следовательно, изображения аватаров должны сохранять качество с углов обзора, отсутствующих в тренировочных данных. Основной вклад данной работы заключается в организации вычислений для достижения производительности в реальном времени. Дополнительно, были исследованы пути повышения визуального качества изображений в приложении с произвольных точек обзора.

В результате, в мобильном приложении baseline модель [36] может вычисляться в реальном времени, вплоть до разрешения 640 *×* 640 пикселей. Для повышения визуального качества аватаров, нейронная сеть получает и выдает данные так, чтобы видимая часть аватара занимала как можно больше места в кадре. Исходный подход обучения нейронной сети был дополнен для предотвращения части визуальных артифактов, и для генерации изображений человека как в полный рост, так и в приближении.

**Ключевые слова:** Вычисления в реальном времени, Мобильное аппаратное обеспечение, Нейронный рендеринг, Глубинная нейронная сеть, Реконструкция аватаров людей, Компьютерная графика.

Научный руководитель:

Имя: Виктор С. Лемпицкий

Ученое звание, степень: к.ф.-м.н.

Должность: доцент

Со-руководитель:

Имя: Ренат М. Баширов

Ученое звание, степень: маг-р

Должность: ведущий инженер-исследователь

3

**Contents**

**1 Introduction 6** 1.1 Computer Graphics state and issues . . . . . . . . . . . . . . . . . . . . . . . . . 6 1.2 Neural Rendering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 1.3 Project tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

**2 Literature Review 12** 2.1 Traditional 3D representations and rendering algorithms . . . . . . . . . . . . . . 12 2.2 Neural Rendering and human avatars . . . . . . . . . . . . . . . . . . . . . . . . . 15 2.3 Metrics of image similarity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19 2.4 Ways of speeding up DNN inference . . . . . . . . . . . . . . . . . . . . . . . . . 21 2.5 Computations on mobile hardware . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**3 Methodology 26** 3.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26 3.2 Mobile application development . . . . . . . . . . . . . . . . . . . . . . . . . . . 31 3.3 Improving quality of synthesized images . . . . . . . . . . . . . . . . . . . . . . . 40

**4 Results and Discussion 54** 4.1 Mobile application results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54 4.2 Training experiments results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59 4.3 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**5 Conclusions 62 Acknowledgements 64**

4

**Author Contribution 64 Abbreviations 65 A Relevant mobile application code 66**

**B Images of the experiments 81** B.1 Auxiliary illustrations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81 B.2 Mobile application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90 B.3 Experiments on DNN training . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

5

**Chapter 1**

**Introduction**

**1.1 Computer Graphics state and issues**

Computer Graphics is a vast discipline of Computer Sciences. In general, it studies theoreti cal methods for synthesis of 2-dimensional (2D) images using computers. It also includes research on computer hardware, used for image computing and displaying. Among applications of com puter graphics there are: typography, cinema, entertainment, scientific visualization and industrial design. From a physical point of view, the 2D images capture information about how light beams pass through the space and reach our eyes. Then a snapshot of all received light is interpreted by our perception as a sight full of shapes and colors.

Images in a computer are usually represented with two major archetypes. The first one – vec tor images [28], that are represented by a set of 2D points, lines, polygonal shapes and mathemati cally defined distribution of color within them. They are more suitable for representing schemes or text, while real world objects would require to define hundreds of thousands of shapes to model all unique uneven patterns of materials and light. Such work is enormous in terms of computations, human hand-craft, and computer file sizes. On the other hand, raster images [31] are used. They are represented by a uniform rectangular grid (referred to as raster), filled with colored squares (referred to as pixels). Typically, colors are defined using Red-Green-Blue (RGB) color model [40], where all possible colors are mixed from 3 primary colors, stored in computers as values in [0; 1] range of real numbers (or similarly as values in [0; 255] range of integer numbers). Although the number of pixels may also surpass millions, the structure of pixel grids is uniform, and each pixel stores the same amount of data, allowing fast encoding and decoding in a computer. Also for most cases, the image data has information redundancy in it, and thus can be compressed [46]. Besides, the pixel grid resembles a canvas from the traditional art. The current computer software for creation of raster images has capabilities similar to real brushes and palettes. This intuitive representation allows to speed up production time for humans. It is known that raster images can be very realistic, proven by the visual quality that can be captured with modern digital cameras [7].

6

Speaking of realism, it is one of the most important challenges in the computer graphics research. The generated images should ideally be indistinguishable from sights we perceive with our own eyes. Unfortunately, the real world is at least 3-dimensional (3D). Thus, it cannot be fully represented with images on a screen or paper, which are 2D by nature. Another obstacle, is that human vision is very sensitive, and it can easily detect unrealistic or anomalous features, e.g. noise, tearing, blurriness and so on. Those are referred to as visual artifacts (also glitches, distortions). However, to this day there have not been discovered a unified metric of realism, that would allow to compare two similar images and tell which will be more plausible for the human perception [98].

Direct usage of 2D images for visualization of the 3D world is not always efficient. For example, if we intend to create an animation (a temporarily coherent sequence of images), it would require to draw the same content with slight adjustment between images. Instead of manually copying the content from one image to another and insuring its coherency and synchronization, it is common to instead approximately model the 3D scene of interest. Such scene representations can take explicit forms, e.g. point clouds, voxel grids, triangular meshes, or even implicit functions, such as signed distance fields (SDF)[92]1. After modeling, at every moment of time the geometry’s appearance can be projected from a certain view point onto an image. This allows to model the scene one time and reuse it for synthesis of dozens of images.

Typically 2D images are synthesized from a 3D scene, using rendering algorithms. They emulate the laws of physics that lie within synthesis of images with physical cameras. One such algorithm is *tracing of rays* that travel starting from a light source, bounce from reflective sur faces, until finally they reach an eye or a camera. This method is one of the most photo-realistic, at least to the state of our theoretical understanding of physical specular phenomena. However, it is exceptionally long to compute in great details. That is why a much more coarse algorithm of *rasterization* is typically used, with each simple geometric part of the scene (e.g. points, triangles, voxels) projected onto a 2D image. The color information for each part is extracted from texture images, that capture physical properties of surfaces, such as color, refraction, reflection, etc (see Figure 1.1). But with either rendering algorithm, there persists a problem that all those proper ties need to be explicitly specified, often by a human, which drastically increases the difficulty of achieving photo-realism. Besides, some 3D scene representations lack details, some are heavy to be efficiently processed, or require enormous amount of computer storage. In either case, an order of magnitude higher number of geometric details needs to be defined compared to just a 2D image.

1See Section 3.1. of [92].

7

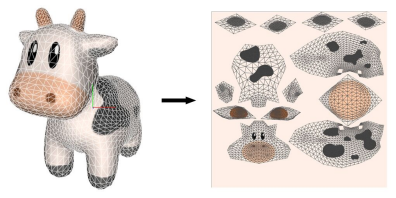


Figure 1.1: Example of a triangular mesh, with a texture image wrapped on it. Each triangle is mapped to texture coordinates, that in turn allow to sample a color patch on the texture. A rasteriazation process projects every triangle onto an image, and interpolates texture coordinates for every image pixel, to then sample corresponding color values from the texture. Picture source: metalbyexample.com/textures-and-samplers.

**1.2 Neural Rendering**

In the last decades, the methods of Artificial Intelligence (AI) and its sub-fields of Machine Learning (ML) and Deep Neural Networks (DNN) have emerged to solve multidisciplinary prob lems of science and business. Instead of modeling a phenomenon algorithmically, they aim to approximate statistical properties of data captured from the phenomenon. Generally, input data is passed down a highly-parameterized computational graph, and output is compared to certain de sired data. During a process referred to as a optimization (training, tuning), the parameters of the graph are iteratively adjusted to better fit the desired data. Afterwards, the computational graph is expected to be general enough to also accurately model the data, that was never seen during train ing. The automation is what makes these approaches groundbreaking. Unlike humans, numerical algorithms can capture very complex statistical correlations or patterns in highly-dimensional data, including images.

A selection of computer graphics problems may also be solved by means of ML and DNN methods. A few notable research topics in this field include:

• image impainting – completion of missing parts on an image with semantically fitting content; • image super-resolution – up-scaling an image and making it more detailed; • reconstruction of 3D objects – automated creation of an object’s 3D representation (see page 7) and materials from 2D images captured with physical cameras;

• neural rendering – automated synthesis of realistic images with certain expected content. 8

This thesis focuses on the topic of neural rendering, although its definition is quite vague. A default approach is to use traditional algorithms to generate a simple image that describes coarse information about a scene, such as general outlines of each separate object, distance from a camera, etc. Then an AI model is used to generate the final image – realistic, detailed, possibly with higher resolution [93]. There is another branch of research, where the AI is used to represent the 3D scene, often implicitly as weights of the AI model. Then this representation is decoded (sampled) into 2D images using deterministic algorithms [66].

It is very rare to have a detailed, realistic and diverse dataset of 3D scenes that we want to reconstruct with neural rendering. Instead, AI models are typically trained on physical camera images, thus combining tasks of 3D object reconstruction and neural rendering. It is known to be a very tough challenge for man-made algorithms, and not as much easier for AI models. On the other hand, there already exist solutions that may surpass any manual human labor on image synthesis or scene modeling [50, 51, 49, 66].

However, training an AI model for generating photo-realistic images is only a half of the challenge. Every AI model is meant to be integrated into a real application and executed (inferred) on some computing device. It can be a stationary computer with lots of computing power. It can also be mobile hardware, since in the last years their computing power became powerful enough to infer AI models in real-time. Such hardware is included in modern smartphones, Augmented Reality (AR), or Virtual Reality (VR) head mounted displays. Running DNN models here directly provides lower latency of computations, better privacy and data security. But also, many perfor mance, compute and power usage constraints may arise:

• AI models that run in real-time applications need to deliver results multiple times per second. • The computing resources are limited due to a small physical size of the hardware. Resources are even further restricted, in order to reduce overheat and to improve battery life. • AI models that utilize customer devices by any means need to deal with compatibility issues (e.g. between operating systems, computing architectures, memory layouts).

This makes real usage of AI models a very challenging task. AI projects have to rely on stan dardized software and hardware solutions, that allow easier integration of new AI advancements. However, a unified way to efficiently run arbitrary AI models on every hardware is absent. It is often, that instead a particular AI architecture is manually integrated for a particular type of hard ware, squeezing the maximum performance out of them. This is the main way-out for the current projects to obtain any business value from combining state-of-the-art hardware and AI.

9

**1.3 Project tasks**

Samsung AI Center at Moscow had been researching on real-time neural rendering of human avatar images. Their recent DNN architecture [36] takes as an input a rasterized image with a body mesh with absent clothing, hair and distinctive person features. When trained with high image resolution, this architecture has proven to fill in those details on the image, producing good full body (FB) images with a single person. However, the performance concerns for real time avatar rendering were not profoundly researched.

During the Skoltech’s Summer Immersion of Master students in 2021, a development of a mobile Android application was started from scratch. It needs to serve as a test of performance, visual quality and compatibility of the aforementioned architecture on mobile devices. The appli cation should prepare input data for the DNN and visualize avatar images on top of camera frames as AR, i.e. as if images are part of the captured environment. Camera movements allow to observe the avatar from arbitrary distance and angle.

The main concern is to obtain a balance of performance and visual quality of the mobile AR application. For instance, with the minimal acceptable avatar resolution of 512 *×* 512 pix els, the performance of the researched DNNs is too low. Each frame computing takes more than 60 milliseconds (ms) on the most recent modern mobile hardware (Qualcomm Snapdragon 888). However, a computation time lower than 33ms per frame is required for smooth AR experience, which is equivalent to more than 30 frames per second (FPS). Moreover, the architecture’s training process could be adjusted with AR usage in mind. For example, when we move camera close to the avatar, effectively only a small part of the body can be seen by the user. But if the DNN generates full body images regardless, a big part of computations is wasted. And the seen part will be in a fairly low resolution. When rendered on a mobile screen with high resolution, it appears as a pixelated image in low details (see Figure 1.2b). Also the DNN should be stable in views that are possible in AR, but not present in the training data, e.g. straight top-down or bottom-up views.

The goal of this thesis is to continue the aforementioned project. The first high-level task is to achieve an admissible trade off of performance and visual quality, when executing the DNN as a part of mobile AR experience. Target devices are the latest mobile phones running on Qualcomm Snapdragon System on the Chip (SoC), with Android operating system. The second task is to research on adjusting DNN training to produce better quality images on zoomed scales (upper body, close-up), while retaining similar quality in full-body scale. However, it is advisable to keep the same baseline [36] DNN architecture if possible, because integration of a completely new pipeline may contradict the implementation of the first task, and will require significant time on obtaining better results and integration.

10

(a) (b)

Figure 1.2: (a) A full-body image generated with the baseline DNN [36]. (b) Full body image synthesis for AR, but only a small image part is visible and in low resolution, the rest is wasted.

The specific tasks are defined in details as follows:

1. To implement real-time input generation on mobile, for the baseline DNNs. This includes inference of a posed mesh of a unified human body model [74], extraction of camera tracking information, mesh rasterization as a 2D image, its passing to the DNN running on mobile.

2. To research on decreasing inference time of the baseline [36] DNN architecture on mobile. It should be possible to infer it with real-time performance of at least 30 FPS, and resolution of the generated images above 256 *×* 256 pixels.

3. To research on DNN’s training improvements, to make it robust to out-of-distribution views. For example, extremely far zooms, extremely close zooms, rotations and angles of view, while preserving full-body quality.

4. To achieve execution of the DNNs on specialized mobile hardware for fast quantized AI computations (see Section 2.5). To compare visual difference between outputs of the default model and outputs on mobile.

5. To research on algorithmic improvements that would compensate weaknesses of plain DNN’s inference, improving either performance or visual quality of avatars.

11

**Chapter 2**

**Literature Review**

This chapter introduces to traditional and Deep Learning approaches of image synthesis, and overviews ways of inferring DNNs in real-time on mobile devices. Sections 2.1 and 2.2 gradually expand on state-of-the-art reconstruction of human images, with more attention to the parts that comprise the basis for this work. The used metrics and optimization losses are described in Sec tion 2.3. Then, methods for decreasing DNNs inference time are given (Section 2.4) with a few peculiarities of the mobile devices targeted by this work (Section 2.5).

**2.1 Traditional 3D representations and rendering algorithms**

Ever since the dawn of Computer Graphics, many primitive representations for 3D world were proposed to reconstruct real objects. The representations may model volumetric properties, such as occupancy, density of matter, radiance, color at a certain point of 3D space; or model surfaces and assign these properties to points on a surface. The definition of the primitives may be explicit or implicit. An explicit surface can be defined by taking every point on a real coordinate plane R2, and mapping it to the third ”height” coordinate explicitly via function *fe*(*.*) *∈* R. An implicit surface is defined by a zero-level set of a multivariable function *fi*(*.*) *∈* R [92].

*Sexplicit* =

( *x*

*y*

*fe*(*x, y*)

! *xy**∈* R2) (2.1)

*Simplicit* =

( *x y*

*z*

!

*∈* R3

*fi*(*x, y, z*) = 0) (2.2)

Unlike surfaces, values of volumetric properties are usually defined only explicitly for all points in the 3D space:

*V* =

(

*fvol*(*x, y, z*)

*xyz*!*∈* R3)(2.3)

Let us look at the common surface and volume representations in details. A *point cloud* is a discrete set of points in R*n*(usually R3). It can model either surfaces (thus each point belongs to

12

the surface) or volumes. The points may be associated with additional attributes, such as: color, density, orientation vector (normal) of a surface. When rendering an image from a point cloud, commonly the points are interpreted as spheres with a certain radius, in order to compensate for sparsity and lack of connectivity between adjacent points [58].

A *polygonal mesh* is the most wide-spread 3D representation, being a compromise between granularity of 3D modeling and computational requirements to render adequate images. It recon structs curvature of a surface using small planar segments, usually triangles or quads, and generally *N*-sided polygons. A set of 3D points (vertices) is defined, along with enumeration of connected vertices. Although each vertex can be associated with additional attributes, using *textures* much finer details can be modeled. Usually, textures are 2D images, that store in the pixels values of attributes (color, surface orientation, etc). Each 3D vertex on a mesh has designated 2D texture coordinates (also referred to as UV-coordinates; See Figure 1.1). A barycentric interpolation of vertices’ texture coordinates is used to sample an attribute value from the texture for a certain point on a polygon. Textures allow to model details, that are too hard to approximate directly with poly gons, e.g. small wrinkles, folds of clothes, complex color patterns, etc. Using textures, a variety of information can be modeled, e.g. a normal direction to an abstract fine-detailed surface for all points on polygons, allowing to calculate realistically looking interaction of light with materials, without additional computational overhead [20].

A *voxel grid* is commonly referred to as a 3D image, for also being a regular grid of prim itives - cubes (voxels) with associated attributes. As volumetric representations, they may store occupation or density of matter at all points in a closed 3D area [35]. Voxel grids are fast to pro cess, but are enormously large in memory. Storing fine-detailed geometry requires shrinking the grid’s scale, but memory consumption grows cubically. Thus, attempts on sparse voxel grids were made, using hyerarchical structures, that are longer to query, but can encapsulate large empty space into very large voxels, and reconstruct fine details using very small voxels where needed [15].

*Implicit functions*, as already described, can model surfaces. For example, we can define a Signed Distance Function for all points in a 3D volume. The values tell the closest distance to the surface, with negative sign if the point is ”inside” a surface. However, to render an image, the sur face (i.e. the zero-distance set) has to be queried repeatedly from the implicit representation. This makes image synthesis computationally expensive, unless an intermediate explicit representation is built to approximate the surface, e.g. using Marching Cubes algorithm [63] to reconstruct a mesh.

The computer graphics evolution led to computing hardware to support some 3D represen tations. For instance, images are now typically rendered using Graphic Processing Units (GPUs), that unlike general Central Processing Units (CPUs) implement operations of linear algebra in a massively parallel way, allowing to process millions of primitives in real-time. Next, we will de

13

scribe the most common rendering algorithms called *ray casting* and *rasterization*. They can be adapted to any 3D representation, although sometimes with too high computational complexity.

For both algorithms, a virtual camera is mathematically modeled. In physical cameras, light that bounces in the 3D space forms an image as it hits the camera’s sensor. Light can be captured from a larger span of view using lenses, that focus light rays into a single spot, referred to as camera origin or focal point. For virtual cameras, a pinhole model is used, where a focal point is chosen, and the 3D scene representation is projected on a pixel grid, as if it was placed at a distance from the focal point, called focal length. From that, a view *frustum* is defined, that is a pyramid that extends from the focal point through the image to infinity, and has apex cut off at the focal length (see Figure 3.1). Objects outside the frustum are invisible to the virtual camera.

The image formation can be mathematically expressed using a projecting transformation **K** (also called intrinsic matrix, see Formula 2.4), parameterized by focal lengths along axes (*fx*, *fy*), image center point (*cx*, *cy*), and axis skew *γ*. This transform maps a 3D point, which is extended to 4D homogeneous space **p** = [*x, y, z,* 1] to a 3D point **pproj** = **K** *·* **p** = [*xp, yp, wp*]. Dividing by *wp*, we get **pimg** = [*xp*/*wp, yp*/*wp,* 1]. The last coordinate can be discarded being equal to 1. Now **pimg** contains pixel coordinates of the projected 3D point. This transform only simulates a camera at the coordinates origin. To place it at any point in 3D space, an additional affine transformation matrix **E** (extrinsic, see Formula 2.5) is introduced, which defines camera rotation **R** and translation **t** in the homogeneous space [92]. The projection is thus **pproj** = **K** *·* **E** *·* **p** .

**K** =



*fx γ cx* 0

0 *fy cy* 0 0 0 1 0



(2.4) **E** =



**R**3*×*3 **t**3*×*1 **0**1*×*3 1



 (2.5)

*Ray casting* method uses the above math to project rays, that pass through all pixels of the image and converge at the focal point. At a point where a surface is hit by the ray, surface attributes are extracted from the 3D representation and the pixel is colored accordingly. To model more real istic lighting, the reflection from the hit-points can be calculated, thus attributes from a secondary hit will be accumulated. This recursive process is also referred to as ray tracing. Such approach is computationally heavy, as 3D representation has to be queried many times for each ray to detect hits. On the bright side, the computational complexity only scales with the number of image pixels, no matter the number of objects in the scene (see Figure 2.1a).

*Rasterization* method is typically used together with mesh 3D representation [31]. It is much more widely used due to its computational efficiency and hardware support. Here, vertices of all polygons are projected onto the image. Pixels that belong to the projected polygons are colored. Additionally for each pixel, the nearest polygon’s depth is stored as depth image (also called *Z*

14

buffer). This helps to resolve occlusion between polygons. Since it is usual that the number of vertices in the view frustum is smaller than the number of pixels, the computation is much faster than ray casting. However, it scales with the number of polygons, requiring to apply algorithms for efficient discarding of occluded polygons in real-time. Also, rasterization loses in producing realistic specular phenomena, such as shadows, since the image formation is not physically-based.

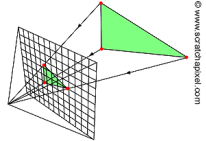
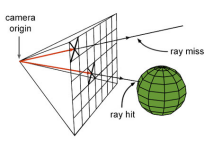
(a) (b)

Figure 2.1: (a) Rays are cast from the camera origin through each pixel, and if a ray hits any surface, the pixel is colored accordingly. (b) In rasterization, vertices of a polygon are projected onto a raster grid, filling the pixels that belong to the projected shape. Images from www.scratchapixel.com.

**2.2 Neural Rendering and human avatars**

As was mentioned in the Section 1.2, any theory that lies inside algorithmic approaches may miss complex correlations in the real world behavior. This can be a corner stone that differentiates realistically-looking and truly photo-realistic images. For this reason, statistical methods of AI (such as DNNs) are applied, to learn a parameterized model from the real observations. In our case, it learns on real photos to synthesize images automatically from descriptive data, such as position of the view in the world, or the desired content in the view. The learning per se comes down to iterative processing of the DNN from input to output, computing loss (fitness) between the output and desired data. The DNN’s weights are updated (optimized) using a gradient value of the loss with respect to every weight.

There are two major modes of DNNs training. The first it to learn on diverse data to under stand different modalities of it. With this knowledge, a feed-forward with novel inputs would yield an acceptable result [33, 73, 50]. Although time of inference is much smaller than of training, it is still rarely real-time, because such generalization requires a substantial number of learnable pa rameters. Another approach is optimization-based. A DNN is trained only on a narrow domain of the data, requiring fewer parameters to fit. Every novel domain needs training of a separate model.

15

Next, we will review relevant advances of DNN research for image synthesis. *Multi-Layer Perceptron* (MLP) is a fully-connected neural network, where layers compute weighted sums of a previous layer, and transform sums by a non-linear function [70]. MLP is known to be a univer sal function approximator [42], yet high accuracy requires lots of parameters. Still, the research community has found it powerful to use MLPs as an implicit 3D scene representation, and then synthesizing the images using deterministic algorithms, such as ray casting [86]. This idea evolved into *neural radience fields* (NeRF, [66]), where MLP maps any scene point (*x, y, z*) and view an gle (*θ, ϕ*) to its color (*r, g, b*) and volume density *σ*. The method can model complex materials and reflections from novel views of static scenes, but takes seconds to query the volume and render an image. It is also suitable for reconstruction of dynamic objects, such as humans[14].

Large MLPs are prone to overfitting, i.e. that they can memorize perfect solutions to the training data and fail to generalize the novel data. *Convolutional Neural Networks* (CNN) rev olutionized [59, 56] image processing with fewer parameters and better generalization. There, a series of 2D convolution operations is applied, that compute weighted sums over input tensors with a small-sized learnable matrix in a sliding-window manner. In MLPs the same data in differ ent places on an image is processed with different weights. CNNs are invariant to spatial shifts, but may have small receptive field - i.e. size of recognized patterns limited by kernel size. Many notable DNNs apply convolutional layers, e.g. AlexNet[56] ResNet[37], VGG[85], U-Net[79].

*Generative Adversarial Networks* (GAN, [33]) is an optimization framework for DNNs that advanced image synthesis a lot since CNNs. In essence, it represents a competing game. A gen erator network *G* learns the data distribution. A discriminator network *D* learns to classify input as either real observation, or produced by *G* (fake). Importantly, the losses for *G* do not need to measure similarity of the real and generated data, it is enough for *G* to minimize accuracy of *D*, by making more realistic fakes. Ever since, many GAN models were proposed [25], the most notable of which is StyleGAN [50, 51, 49] that is able to generate artificial high-resolution images of hu man faces, but far from real-time inference and rich articulation. Apart from that, GANs can be a part of architecture to generate any data at training time and optimize with respect to it [36, 11].

Speaking of DNN optimization, it is desirable to prevent overfitting and at the same time speed up convergence of results to the optimum (escaping underfitting). Overfitting can be pre vented with more diverse real data, or by making the existing data diverse via augmenting, e.g. replicating data with varying color, erased content, affine transformations. Also, regularization of weights can be applied to make training harder, e.g. dropout[88] – random nullifying of activa tion values; or weight decay[64] – small decrease of weights towards 0 absolute value on every optimization step.

16

The reason of underfitting can be a deficit of number of parameters to fit the data. Also, a covariate shift can be frequently observed, e.g. training and novel (test) data may have slightly different distributions, or even activations of internal layers may have different distributions after updating weights of a network. To compensate the shift, normalization techniques are applied. Batch Normalization (BN, [47]) algorithm during training normalizes intermediate activations to the mean and variance of the whole batch of data, see Formula 2.6. Here *γ* and *β* are learned vectors of same size as number of input channels. Besides, the batch statistics are accumulated on each step with exponential averaging, so that during inference batches of any size could be processed. Instance Normalization (IN, [96]) follows the same formula, but the statistics are computed over each sample, rather than a batch of samples. These and other normalization schemes may increase

DNNs convergence speed and reduce sensitivity to hyperparameters.

*y* =*x −* E[*x*]

~~p~~Var[*x*] + *ϵ∗ γ* + *β* (2.6)

There is a branch of research called *deferred neural rendering*, where 3D meshes of objects are rendered with a neural texture – a tensor, that unlike usual textures with 3 color channels (RGB), has generally *C* channels [93]. A DNN is trained to map this neural image to a colorful RGB image with segmentation (RGBS/RGBA), see Figure 2.2b. The motivation is to give a clue to the DNN on what to synthesize, and to optimize the neural texture to implicitly model specular properties of real materials. Also a coarse mesh of the object can be used, that does not model small details, and the DNN will have to put information into the neural texture about reconstruction of these details [36, 78]. Crucially, the algorithm that renders the mesh with the neural texture has to be differentiable [52], to optimize the texture.

This thesis focuses on human avatars neural rendering, i.e. recovering of photo-realistic ap pearance, and modeling movements of limbs, face, hair, clothes. The reconstruction may cover the full-body (FB) [104, 36, 78], upper-body (UB) [24], or only face of a person [61, 11]. The source data may vary, e.g. a single frontal photo (one-shot avatar), multiple photos from sides (few-shot avatar), a video sequence (video-based avatar). Either can be captured on a monocular camera[36], stereo camera[102], multiple cameras[61, 84]; with or without camera calibration parameters (in trinsics as in Formula 2.4). Images alone impose depth ambiguity, i.e. the true size of objects on an image is unknown. Depending on proximity to the camera, they appear smaller or bigger. It can be resolved with depth sensor information [97], or by scanning 3D meshes of objects [14], both requiring expensive equipment. Thus, it is desired to make realistic avatars from common data in-the-wild – a few images or videos taken on a monocular uncalibrated camera. The task is ill-posed and combining quality with real-time inference is challenging. That said, photo-realism is not achieved yet.

17

We follow StylePeople [36] architecture as a baseline for this thesis. It is based on deferred neural rendering, as described above (see Figure 2.2b). The input of a neural renderer network is a rasterized image of a generic human body mesh, that does not contain any personal details, such as facial features, hair, clothes. It is rasterized with a neural texture tensor, that has usually between 4 and 16 channels. Feeding this ”neural” image through the renderer, the personal features are filled in, and a colorful image is returned, along with body contour segmentation. Both neural renderer and neural texture are jointly optimized using image-based losses (more details in Section 2.3), including an adversarial loss using a discriminator network (see the paragraph on GANs above).

The neural renderer’s architecture is a CNN called U-Net [79], consisting of an encoder and decoder part. In encoder, input is gradually processed with convolutional down-scaling filters. Next, passing through the decoder, the data and the corresponding encoder layer’s activations are concatenated and further processed with convolutions, gradually up-scaling to the original resolu tion. The U-Net’s output is fed through two independent CNNs, predicting color and segmentation images that make up the final RGBA image. Commonly to improve training, the encoder part is replaced with a powerful CNN backbone, in our case ResNet[23, 37]. The discriminator’s archi tecture is exactly as in StyleGANv2 [51], the details of which are irrelevant for this work.

The generic human body model used in StylePeople is *Skinned Multi-Person Linear Model - Expressive* (SMPL-X [74], derived from [62, 21]). It allows representing a variety of human bodies, including expressive faces and hands as 3D triangular meshes with *N* = 10475 vertices, and *K* = 54 joints for limbs, face and fingers (See Figure 2.2a). It does so by learning bodies distribution from many 3D scans of people. A particular body shape is derived as a linear combination of learned body blend shapes *S* = [*S*1*, ..., S|β|*] *∈* R3*N×|β|*, where *β* is a vector of linear coefficients. The body blend shapes are orthonormal principle components of the learned mesh vertex displacements. Typically, about 10 body shape components are enough for reconstruction of any body constitution. Face vertices have their own blend shapes *E* = [*E*1*, ..., E|ψ|*], combined with *ψ* coefficients, similarly about 10 are usually enough.

Body mesh can be posed using learned orthonormal components*P* = [*P*1*, ..., P*9*K*] *∈* R3*N×*9*K* of vertex displacement with respect to a pose. The pose is defined by axis aligned rotations of body joints *θ* (*|θ|* = 3*K*). It is then expanded into rotation matrices *K ×* 3 *×* 3 using Rodrigues formula [77], notated further as *R* : R*|θ| →* R9*K*. The computed body shape offsets, face shape offsets and pose offsets are simply added to an average body mesh *T*¯ in a default pose *θ∗*. Formally, a posed triangular mesh *Tp* can be obtained as:

*Tp*(*β, θ, ψ*) = *T*¯ + *BS*(*β*; *S*) + *BE*(*ψ*; *E*) + *BP* (*θ*;*P*) (2.7)

*BS*(*β*; *S*) = X*|β| n*=1

*|ψ|*

*βnSn, BE*(*ψ*; *E*) = X *n*=1

*ψnEn BP* (*θ*;*P*) = X

9*K*

*n*=1

18

(*R*(*θ*)*n − R*(*θ∗*)*n*) *Pn* (2.8)

SMPL-X also solves issues of mesh deformations and self-intersections, e.g. in poses with bent arms or rotated neck. However, it is computationally expensive to compute and may be skipped, since the neural rendering may handle these artifacts. SMPL-X also proposes an algorithm for optimization-based fitting of the body model to a single image, which is used in StylePeople too.

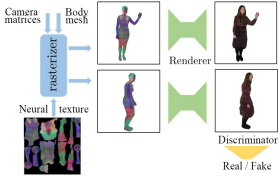
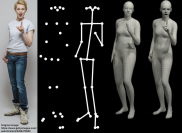
(a) (b)

Figure 2.2: (a) SMPL-X body model results. Joints are predicted in image space, then body shape and 3D pose are fit to maximize alignment with the source image. The final mesh is generic, lacking person-specific features as hair and clothes. (b) StylePeople data flow. SMPL-X body mesh is rasterized with the neural texture, the image is then refined by a neural renderer, to fill in the person’s appearance, clothes, hair. A discriminator network adds supervision to the renderer, by classifying generated and real images of the person. Images adapted from [74] and [36].

**2.3 Metrics of image similarity**

For comparison of real and synthesized images, many metrics can be applied. Those that are differentiable and fast to compute, are usually used as training losses, by minimizing which the DNN learns how to generate images similar to the real. The next description will follow the notation: **f** - a fake synthesized image, **r** - a real image, **f***segm*/**r***segm* - their segmentation masks, **Af**/**Ar** - intermediate DNN layers activations when processing fake and real images respectively.

*Mean Absolute Error* (MAE or L1 [89], ideal value 0) measures absolute differences between corresponding pixels. It simplifies the training process, by pushing pixels to equality, but does not capture many image degradation effects like blur, warping, etc:

MAE(**f***,* **r**) = 1*N*X*N i*=1

*|***f***i −* **r***i| ∈* [0*,* +*∞*] (2.9)

*Mean Squared Error* (MSE or L2 [89], ideal value 0) measures squared differences between corresponding pixels. In terms of image comparison, similar to MAE. However, it has smoother values distribution, and penalizes high differences much more. Sometimes such penalization is too

19

harsh, even though perceptually the difference is very small.

MSE(**f***,* **r**) = 1*N*X*N i*=1

(**f***i −* **r***i*)2 *∈* [0*,* +*∞*] (2.10)

*Learned Perceptual Image Patch Similarity* (LPIPS [107], ideal value 0) is based on an ob servation, that image-based neural networks (commonly VGG [85], Pix2Pix [48], AlexNet [56]) tend to learn inside their weights structured patterns from images. Given a new image, activations of the network will contain a proportion of those patterns that can be found on the image. Thus, we could infer such network with fake and real images, and compare activations. If they are identical, then the original images are perceptually similar on a global level. The difference in activations is applied as a loss during training. The metric is computed as an average MSE distance between corresponding activations (also normalized by the tensors size to equalize contribution of deep layers).

LPIPS(**f***,* **r**) = 1*L*X*L l*=1

1

size(**Af***l*)MSE(**Af***l,* **Ar***l*) *∈* [0*,* +*∞*]*, L* is number of VGG layers (2.11)

*Structural Similarity* (SSIM [99], ideal value 1) metric is based on an idea, that perception of noise on an image is dependent on luminance (mean of image values *µ*), contrast (standard deviation of image values *σ*) and structure (covariance between images *σxy*). The metric is computed for two images from components, that measure luminance difference (*l*(**f***,* **r**)), contrast difference (*c*(**f***,* **r**)), structure difference (*s*(**f***,* **r**))). Importance of components is also controlled by *α*, *β*, *γ* powers.

*l*(**f***,* **r**) = 2*µ***f***µ***r** + *C*1

*µ***f**2 + *µ***r**2 + *C*1*c*(**f***,* **r**) = 2*σ***f***σ***r** + *C*2

*σ***f**2 + *σ***r**2 + *C*2*s*(**f***,* **r**) = *σ***fr** + *C*2/2

*σ***f***σ***r** + *C*2/2

SSIM(**f***,* **r**) = [*l*(**f***,* **r**)]*α·* [*c*(**f***,* **r**)]*β·* [*s*(**f***,* **r**)]*γ ∈* [*−*1; 1]

(2.12)

where *C*1 = (*K*1*L*)2, *C*2 = (*K*2*L*)2, *L* is a range of values stored in pixel channels (usually it is 1 for channels in floating-point format, or 255 for integer 8-bit channel format), *K*1 and *K*2 are arbitrary constants to ensure numerical stability for near-zero denominator. SSIM is usually computed in a 11*×*11 sliding window and averaged over the whole image. It can detect degradation due to noise, but might be insensitive to blur and color shift.

*Multi-Scale Structural Similarity* (MS-SSIM [100], ideal value 1) is an extension of SSIM, but contrast and structure terms are computed on multiple scales, i.e. the original images are pro cessed with a low-pass filter and downsampled by a factor of 2, a total of *M* times. All the computed multi-scale components are multiplied to the metric. The metric was shown to be the same or better than SSIM in detecting many types of distortions:

MS-SSIM(**f***,* **r**) = [*l*(**f***,* **r**)]*α·*Y*M j*=1

[*cj* (**f***,* **r**)]*βj*[*sj* (**f***,* **r**)]*γj ∈* [*−*1; 1] (2.13)

*Peak Signal-to-Noise ratio* (PSNR [41], ideal value +*∞*) metric comes from signal process ing theory. It captures a ratio of the maximum value in an noise-free signal to the average noise

20

magnitude (can be captured by MSE). The metric indeed helps to detect noise levels, but it is much less useful as image similarity measure than other metrics, such as SSIM.

PSNR(**f***,* **r**) = 20 *·* log10 (max(**r**)) *−* 10 *·* log10 (MSE(**f***,* **r**)) *∈* R (2.14) *Adversarial* loss (GAN-loss [33], ideal value 0.5) is used as a part of Generator - Discrimi nator DNN training architecture. It measures how well the discriminator learned to distinguish real images from fake (by assigning probabilities of realism: 0 is fake, 1 is real). The generator’s task is to synthesize images that cannot be detected by the discriminator. Ideally, discriminator should report probability 0.5 on average, indicating that it essentially reports a random guess.

*Feature matching* loss (FM [81], ideal value 0) is analogous to LPIPS, but the intermediate activations on fake and real images are taken not from external image network, but from discrimi nator network. The motivation is that the generator network will learn to produce images that are perceived as real by the discriminator.

*Dice* loss (ideal value 1, [90]) is widely used in various pixel segmentation tasks to learn segmentation masks. It maximizes area of masks intersection and minimizes union of the areas (to prevent false positives in the predicted mask). The formula treats masks as sets of pixels where

segmentation value is greater than 0.

Dice(**f***segm,* **r***segm*) = 2*|***f***segm ∩* **r***segm|*

*|***f***segm|* + *|***r***segm|*(2.15)

**2.4 Ways of speeding up DNN inference**

State-of-the-art DNN methods indeed achieve high quality results, but may also have tremen dous computational costs. As an example, the models of natural language processing may surpass billions of trained parameters (weights): Turing-NLG has 17 billion parameters [80], GPT-3 - 175 billion [22], Megatron-Turing NGL - 530 billion [87]. This number affects the amount of required computer memory, as all the parameters, computed values (activations) of intermediate DNN lay ers, and analytic gradients need to be stored. Secondly, sufficient computing power is needed to make training time more feasible. For instance, a computing cluster of 10000 GPUs, 285000 CPU cores was built in order to train GPT-3, still requiring more than a month of constant training [12].

After training, the amount of minimum computing power to perform DNN inference is gen erally a lot smaller, because we can load and compute neural layers one by one without storing intermediate activations. Nevertheless, inference may still require dozens of GPUs, a handful of memory and seconds to compute. Thus, a few research directions of Deep Learning are dedicated to reduce the required computing power, trying to preserve the similar model’s quality.

21

One such approach is known as *knowledge distillation*. Its idea, is that the learned values of DNN’s parameters are not equivalent to knowledge of how to perform a task. In fact, by knowl edge we can consider any sufficiently good mapping from input data to the desired output data [39]. Thus, we could try to train the original model fully to yield great results, and then to train an other DNN ”student” model with much fewer parameters. Both models are inferred with the same input data, and the student model is supervised to output the same results as the original model. Mimicking intermediate activations of the original model is also an option to replicate even more knowledge. The simplified neural architecture of the student models helps to reduce computation time and overfitting. That is because with fewer parameters there is less capacity in the student to remember the training data, and also imperfections of the original model’s predictions additionally regularize the student [34, 10]. However, knowledge distillation adds up more research work, with experimentation on neural architectures of the student, the training routines and hyper-parameters.

There is an adjacent group of methods, called *neural architecture search*. It aims to automat ically find a DNN architecture that would outperform hand-crafted architectures on the given tasks. The motivation is to both reduce the committal effect of selecting a sub-optimal neural architecture by researchers, and to also reduce the amount of parameters required to accomplish the same tasks [29, 75]. However, a single search step requires to train the current network, collect quality metrics and then update the architecture. The convergence may require dozens of steps. This limits neural architecture search to lightweight DNNs, and it is hard to apply for big neural architectures.

Another option comes from an observation, that some activation values of DNNs may be less important to the overall output than the others. Such activation values usually have low variance with respect to different input data, i.e. they frequently yield similar values. The corresponding learned parameters could then be replaced in the architecture by constants, if the quality does not decrease much. Such approach is called *pruning*. Similarly to knowledge distillation, it allows to train a big model with the full capacity, which eases training. Then the obtained knowledge is squeezed to the lower number of parameters [60, 19, 30].

On a computer hardware level, DNNs are usually trained and inferred with weights and ac tivations represented with 16-bit/32-bit floating point numbers (also referred to as half-float or FP16; full-float or FP32 respectively). A *quantization* approach switches to integer number for mats, with lower memory usage, e.g. 4-bit/8-bit/16-bit integer numbers (INT4/INT8/INT16). The main motivation is that integer operations can be computed faster than floating-point ones, their physical circuits are smaller, thus decreasing device’s size, overheating, energy consumption [71]. The quantization approach relies on a fact, that a vector of floating-point numbers *x* can be approx imately expressed as

*x ≈ s′x· x′***int** = *sx ·* (*x***int** *− zx*) (2.16)

22

where *s′x*is a floating-point scalar, *x′***int** contains only integer values. Then they can be reformulated using an integer zero-offset *zx*, so that values in *x***int** are mapped uniformly to range (0*,* 2*b −* 1), with *b* being bit-width of a number. Having a floating-point vector *x*, it is quantized as: *x***int** = round(*x*/*sx*) + *zx* (2.17)

Since many operations in neural networks can be computed using dot products of vectors, this allows to represent them in a quantized form. For example, a basic operation of matrix-vector mul tiplication with a floating-point weights matrix *Wn×m* and a bias vector *bn*. The output activation vector *An* could be computed from an input activation vector *xm* as:

*A* = *b* + *W x*

*≈ b* + [*sw*(*W*int *− zw*)] *·* [*sx*(*x*int *− zx*)] = *b* + *swsx*(*W*int*x*int *− x*int*zw−W*int*zx* + *zxzw*)*.*

(2.18)

The last two terms shown in blue can be pre-computed for the whole DNN, and merged with the bias vector *b* [69]. However, since there is still an extra overhead from computation of *x*int*zw*, specifically for weights it is common to use symmetric quantization, where floating-point values are mapped to (*−*2*b−*1*,* 2*b−*1 *−* 1) integer range, with floating-point 0 mapped precisely to integer 0, thus making *zw* = 0, and eliminating the overhead.

Typically, a post training quantization (PTQ) is applied to weights and activations of a DNN, because a complete experiment can be quantized via a provided software tool, and automatically be sped up from quantized computations. On the other hand, if precision of the outputs is a priority, a quantization-aware training (QAT) [55] approach is used. Here architecture tweaks are required to do quantized inference and weights updates, with an idea to make the model learn to deal with the quantization imprecision. As a downside, it enlarges training time and adds issues with analytic gradients back-propagation through rounded values [18].

Usually, the uniform quantization is appealing for simplicity of hardware implementation. However, the quantization range defined by constants *s* and *z* may be selected too wide, e.g. min max range of values in a tensor with existing outliers. Thus, a lot of precision will be lost as different numbers will be mapped to the same integer value. Instead, the range can be selected to minimize MSE with respect to the original floating-point values [16, 76]. If hardware supports it, instead of defining per-tensor quantization constants, they can be defined per-channel for more precision. Alternatively, a Cross Layer Equalization [67] can be applied to re-parameterize quantized weights and biases, compensating the channels’ discrepancy of ranges, without adding runtime overhead. A non-uniform quantization can also be used to devote more precision to near mean values, and less to the outliers, but the hardware support of it is very limited [17].

Moving on, DNNs also suffer from computational inefficiency, when similar input data leads to computing identical activations of layers. The DNN’s architecture can be designed to cache

23

and reuse these values. It can be applied to video stream processing or synthesis, where multiple consecutive frames have similarity [53]. Additionally, filters of CNNs may frequently contain repeated weights after being quantized. Thus, as a filter slides over the data, certain dot-products can be reused [38]. While having a potential to speed up a certain DNN architecture, it requires immense manual hand-craft, which slows down active research and experimentation.

The last but not least appoach, is to design the neural architectures efficiently, i.e with a good trade off of inference speed and outputs accuracy. It is obtained by reducing the number of parame ters, applying only fast-to-compute neural layers, and composing them into a sparse computational graph. It leads to a better utilization of memory caches and computing cores of the hardware. Often, such models are implemented using low-level parallelized code, that increases the number of floating-point operations per second (FLOPs) even further. The notable examples are: Mo bileNetV1 [44], MobileNetV2 [82], MobileNetV3 [43], ShuffleNetV1 [108], ShuffleNetV2 [65], CondenseNetV1 [45], CondenseNetV2 [105], EfficientNet [91]. Although the original papers ob tain similar or better results with their proposed architectures, the drop-in replacement is not guaran teed for every task. In this work a significant drop of quality is observed, when a baseline ResNet18 backbone was replaced with either MobileNetV2 or EfficientNet.

**2.5 Computations on mobile hardware**

Traditional desktop computers have hardware, where components are designed to be inde pendent and replaceable. This is achieved through standardization of connectors and signaling. This design is convenient when physical size of devices, power usage, or necessity of active cool ing are not issues. On the other hand there is mobile hardware, which can be found in modern smartphones, watches, AR and VR devices. There exists a strong desire to minimize physical size, weight, battery power usage and heat generation. Despite the wide variety of mobile devices and vendors, the state-of-the-art mobile architectures look similar on a high level. They are presented as Systems-on-a-chip (SoC), also referred to as heterogeneous platforms, where computing de vices of different underlying principles are fused under a single silicon. The computing work is offloaded to the designated computing devices, to minimize idleness and latency. Such platforms may accelerate general computing (CPU), graphics computing (GPU), signal processing, network ing, cryptography, external devices, camera processing, AI computing, sensing, etc. To name a few SoC platforms, there are: Nvidia Tegra [3], Samsung Exynos [6], Qualcomm Snapdragon [5]. Notably, all of them advance towards providing special computing devices for AI, and DNNs in particular [26]. Such AI accelerators can come by the names: Deep Learning Accelerator, Neural

24

Processing Unit (NPU), Tensor Processing Unit (TPU), etc [101]. This work by default targets devices running Qualcomm Snapdragon SoC.

The latest chip called Qualcomm Snapdragon 888 consists of an 8-core CPU, a GPU with mixed precision AI inference support (vectorized half-float/full-float), a sensing processor, and a Digital Signal Processor (DSP). The latter is especially optimized for parallel integer computations, which are common in real-time 1D or 2D digital signal processing (e.g. Fast Fourier Transform algorithm). It supports parallel scalar, vector and tensor operations, while retaining state-of-the-art low level of power usage. Besides, it is possible to use DSP as an AI accelerator to infer a quantized DNN, and get massive benefit in inference speed and energy consumption. All accelerators of the SoC are located on the same chip and have access to shared memory, allowing to reduce overhead of data transfer (which is common in desktop systems). The claimed performance of the combined accelerators is 26 TOPS (Trillion Operations per Second). The platform also offers a triple-core chip for multithreaded camera processing with up to 2.7 Gigapixels per second of throughput [95].

Importantly for this work, Qualcomm Snapdragon’s DSP works with tensors layout called BHWC (Batch-Height-Width-Channel), that is, the data of a single spatial element (pixel) is stored contiguously (e.g. *C* values for a pixel 1 are followed by *C* values for a pixel 2, and so on). This will be utilized during real-time input data generation for the DNN researched in this work. The other possible format is BCHW (Batch-Channel-Height-Width), which is usually used in software libraries for DNN training on desktop computers.

As well as usual quantization approaches described in Section 2.4, for Snapdragon DSP a data-free quantization algorithm is implemented, which does not require estimating quantization ranges from example input dataset [68]. It features high quantization speed, which used to be critical with big datasets, and also achieves near-identical outputs’ quality for quantized DNNs.

An AI model can possibly provide software level improvements for better hardware utiliza tion. For example, the work [72] suggests a pipeline approach, where a model is split into several stages. Instead of processing input from start to finish, each stage receives new input as soon as it finished the previous calculation. By combining usage of multiple accelerating devices on mobile hardware, it is possible to almost double real time performance.

25

**Chapter 3**

**Methodology**

This chapter describes implementation of the mobile Augmented Reality (AR) pipeline of avatars rendering, and experiments on increasing the DNN’s [36] images quality for AR scenario. Section 3.1 lists target hardware and software, including the used libraries, tools, and configuration of the baseline model. Section 3.2 describes the mobile pipeline and algorithmic tricks used to achieve real-time performance. Section 3.3 presents motivation and implementation details for experiments with the DNN’s training procedure.

**3.1 Experimental setup**

The baseline DNN [36] model can be trained on a monocular video sequence of frames with a single person shown from multiple sides. There are several pre-made training video sequences with different people at our disposal. They were captured for approximately 1 minute on a station ary monocular smartphone camera, in resolution 1920 *×* 1080 pixels. Only each 8th frame from the video was retained to get a representative set of distinct body poses. Then, for each frame a segmentation mask was predicted using an external pretrained segmentation model Graphonomy [32], to replace color of background objects with 0 values. Next, SMPL-X [74] body model was fit to all the frames, i.e. predicting pose vector *θ*, facial expression vector *ψ*, and body shape vector *β*. The latter is purposely predicted to be the same across all the frames, in order to bias the DNN to synthesize images for this body shape only, which will be exploited in real-time inference on mobile in Section 3.2. Using body model fits, a posed body mesh for each frame can be obtained as was described in Section 2.2.

Sometimes, 3D model fits may be misaligned with complicated ground truth 2D frames, e.g. due to complex twists and bending of joints, occluded body parts. To reduce the effect of training the DNN on outlying unstable inputs, many of the falsely fit ground truth images were manually filtered out from the training set. It was observed, that the quality of body model fits is critical for the avatars final quality. Hence, for the scope of this thesis, we decided to take a single training sequence, that has the best body model fits we could obtain. The training pipeline adjustments

26

discussed in Section 3.3 will be primarily applied to this training sequence. This also allows the fair quantitative and qualitative comparison. For simplicity, when we further mention *the training sequence*, we mean this single training sequence. A few promising experiments will be repeated with a few other sequences to verify stability of results (for example, see Figure B.33).

After filtering out the bad frames, the training sequence contains 410 samples. Out of them we holdout 13 frames as a validation set. It includes 11 consecutive frames from a frontal view, in poses that do not resemble any other training data; and also 2 frames with side and back view. As for a test set, we made a sequence of continuous difficult poses and camera angles. It includes an animation with extensive hands gesturing, a view on the avatar from top, bottom, far away, and close-up to the face. The test sequence is completely unrelated to the training data and was designed by us to verify that the DNN is capable of rendering frames that may be encountered by a user in an AR session. The possible visual artifacts may include flickering between two similar frames, clothes shaking, color shifts, blurriness, etc. See Appendix Section B.3, featuring a few training and test images that we use. Lacking an absolute metric of visual quality, in this research we mainly rely on manual inspection and comparison of the test video sequences.

In the next paragraphs we will elaborate more on the specific configuration of the baseline DNN architecture, that was first mentioned in Section 2.2. By default, we stick to 512*×*512 pixels resolution of synthesized images. Both the ResNet18 encoder and decoder of the neural renderer consist of 5 levels of 2-layer CNNs with stride 2 and kernel sizes either 3 *×* 3 or 5 *×* 5. The U Net yields a tensor with 64 channels and the original spatial resolution. Two independent CNNs with 2 layers each (also called heads) then map it into a 3-channel RGB image, and a 1-channel segmentation mask, that are concatenated into the final RGBA avatar image.

The neural texture is a tensor of 16 channels and resolution 512 *×* 512 by default (although may be arbitrary). For the SMPL-X body model mesh, there is a known mapping from 3D mesh vertices to the same 2D texture image coordinates (as on Figure 1.1), regardless of body shape (tall, short, slim or obese). In addition to the original resolution, extra texture scales of size 16 *×* 2*k ×* 2*k, ∀k ∈ {*3*, ..,* 8*}* are created and jointly optimized. This set of scales is referred to as a multi-scale neural texture. Before rasterizing the body mesh, all the scales are upsampled to 512 resolution and averaged into a single texture. The multi-scale texture allows to share information among adjacent vertices and proven to reduce flickering of the output avatar images. Also, the baseline neural texture uses a special initialization. It is computed as spectral coordinates of vertices on the graph, that is made out of the body mesh. In short, the graph’s Laplacian matrix is calculated, and Eigenvectors of it are called spectral coordinates [106]. A few first dimensions of these vectors are used to initialize the neural texture. The purpose of such initialization is in separation between different parts of the body (see Figure B.1), e.g. there are channels that assign opposite values to

27

left/right limbs, or lower/upper body, or adjacent fingers and so on. Such initialization helps to guide the neural renderer to uniquely distinguish the body parts, improving convergence speed.

During training, the synthesized image and its segmentation mask are supervised with a weighted sum of training losses. By default Dice, L1, LPIPS (VGG [85] based), Adversarial and Feature Matching losses are used, with weights 25, 25, 20, 1, 1 respectively. The adversarial loss is a Binary Cross Entropy (BCE) loss of predictions made by an ensemble of *N* = 3 discriminator networks. Each discriminator is a CNN that gradually decreases the spatial resolution and increases the number of channels, until finally a Sigmoid function returns a probability in range [0; 1] of the image being real. The discriminators are supervised to improve their accuracy on both synthesized and real images, while the neural renderer is supervised to decrease discriminators’ accuracy for the synthesized ones. See Formula 3.1 for BCE, when computing it for a real image *x*, then *y* is set 1 for discriminators; if *x* is a generated image, then *y* is 0 for discriminators, but 1 for the neural

renderer.

*BCE* = *−*1*N*X*N i*=0

*y ·* log2(*Di*(*x*)) + (1 *− y*) *·* log2(1 *− Di*(*x*)) (3.1)

Adam [54] optimizer is utilized for training of the neural renderer, neural texture and dis criminators. They are optimized with learning rate (LR) of 10*−*3, 10*−*3and 10*−*5respectively. By default, the Adam’s first and second momentum values are 0.9 and 0.999 respectively, without weight decay (also known as L2 penalty on norm of the optimized parameters).

In the baseline training process, a batch of random training samples is selected. Using body model’s mesh, a bounding box is found that tightly encloses the person’s extents on the ground truth image. The image is cropped by this bounding box using bilinear interpolation to get a 512 *×* 512 image with the person’s body in the image center. With the same resolution and alignment, the body mesh is rasterized. Then, both the ground truth and rasterization are processed with image-space augmentations (see Figure B.12), that perform random affine rotation, scale and translation of the image to improve generalization of the neural renderer to different camera views on the same avatar. Then the neural renderer is infered with input rasterization, and loss is computed between real and synthesized images. The gradient is computed and applied to all the learned parameters (the neural texture, neural renderer and discriminators) on every single batch. We train the pipeline on desktop hardware – one GPU NVIDIA Tesla P40. It takes about 16 hours to complete a single experiment of 500 epochs on about 500 frames of ground truth images, with batch size 8 and resolution 512*×*512 pixels. All training computations are carried out in full-float FP32 numbers format. As validation metrics, we compute LPIPS (AlexNet [56] based), SSIM, MS-SSIM, and PSNR, as well as tracking values of L1, Adversarial and Dice losses on the validation set.

28

The experimentation on deep neural networks is carried out in Python programming language and PyTorch framework for the DNNs training. We use many Python libraries for data processing and visualization: NumPy, Matplotlib, OpenCV, Pillow, Open3D. In order to optimize parameters of the neural texture, it needs to be rasterized with the body mesh using differentiable rendering algorithm, we use Nvdiffrast library for that [57]. The image-space augmentations need to be differentiable for the same reason, for which we utilize Kornia library [27].

The primary mobile hardware that we target with our mobile application is SoC Qualcomm Snapdragon 888, which is one of the latest high-end platforms that features a fast and power effi cient DSP computing unit for AI, and high throughput for processing mobile camera data in real time. Although the preceding versions of the platform also contain the DSP, they are too slow to infer the baseline DNN in real-time. Qualcomm Snapdragon 835 is the oldest platform that we succeeded to launch the mobile application on, although with unacceptable performance.

We develop the mobile application in Java and Kotlin programming languages. The devel opment environment consist of the Java Software Development Kit (SDK) and Android SDK. We perform installation of the compiled code on a mobile device via Android Debug Bridge (ADB) software utility. In order to launch the PyTorch DNN on mobile, first of all we use Snapdragon Neural Processing Engine SDK (SNPE, [1]). It includes Android runtime libraries, as well as desk top software tools for conversion of DNN architectures into binary files, that can be loaded onto mobile devices and executed. Among the tools there is a utility for post-training quantization of a neural architecture. We supply it with 5-10 input frames from the test sequence. The neural ren derer network is inferred with those frames, and the range of quantization is computed for each intermediate tensor. Then all the weights are quantized as was described in Section 2.4. The di rect conversion of PyTorch DNN models into SNPE format is not supported, however there is an Open Neural Network Exchange format (ONNX, [2]), which supports conversion from PyTorch to ONNX (provided by PyTorch), and from ONNX to SNPE (provided by SNPE SDK). The final file with the quantized DNN is passed along with the quantization constants to the mobile application to accordingly generate quantized input frames.

On the mobile application’s side, we use SNPE Android Runtime library to load the model’s file and to specify performance profiles. Those control how much of computing power should be devoted by the device. We use ”Sustained High Performance” profile, since we intend to continu ously infer the DNN over a long period of time with low latency. On the other hand, there is the absolute highest performance profile, that would lead to the best performance over a short period of time, that would drastically decrease upon reaching the overheating limit. Prior to inference, SNPE should be given with pointers to memory locations, where to read the input data, and where

29

to write the output data. We accordingly preallocate buffers on CPU and GPU, and use them for the whole run of the mobile application to minimize costs of memory allocation on each frame.

For the DNN inference, we need to generate an input frame (see Figure 2.2b), that would show the body mesh oriented as an object of Augmented Reality. Hence, the orientation should correspond to the mobile device’s camera. This requires continuous tracking of camera position in the real world, as well as positions of the surrounding environment. We use ARCore SDK [8] that provides it with little programming required. It uses inertial measurement units, camera images and depth sensor of the device to keep track of distinct points in the camera view. This allows to estimate proximity of the real world objects, and to reconstruct vertical and horizontal flat surfaces, such as floors, desks, walls. The location and bounds of these surfaces can be queried from the ARCore. It also allows to place and track anchor points on the detected surfaces. The tracking information can be retrieved from ARCore in a form of transformation matrices of the anchors and camera (extrinsics, as in Formula 2.5). The projection matrix of the camera (intrinsics, as in Formula 2.4) can also be retrieved, allowing to render images that span the whole screen of the device.

The generation of input frames is in essence a classic rasterization of a triangular mesh onto an image using a 2D texture. This process can be delegated to the high-performance graphics library, which is called Open Graphics Library (OpenGL, [9]). It provides low-level control of a GPU device, to layout and are pre-process the data of meshes and textures. Then, using GPU programs written in OpenGL Shading Language (GLSL, [4]), also referred to as shader programs, we can make linear algebra computations on individual vertices of the mesh (vertex shader), or individual pixels on the rendered image (fragment shader). Since all those primitives can be processed inde pendently, this computation is automatically parallelized on GPU by OpenGL, thus achieving very high performance of rasterization. We also use it to render camera frames on the screen. On top of them we render the avatar images, resulted from the DNN’s inference (see Figure B.16). In fact, we use a subset of the library, called OpenGL for Embedded Systems (OpenGL ES) version 3.2, that consists of core functionality for rasterization, lacking many high-end graphical extensions.

The application is compiled with all performance optimizations enabled. The frame time of the mobile AR experience is measured directly in the application. We report timings of processing steps of the current frame via logging, namely: time of SMPL-X body mesh inference from a pose vector, time of rendering the DNN’s input frame on GPU, time of delay between obtaining this data and receiving it on the DNN’s side, time of DNN’s inference, time of rendering the inferred outputs on the screen. However we notice, that by far the highest processing time is taken by the DNN inference step. To measure its timings more accurately, we created a separate mode for the application, which we call ”benchmarking”. In this mode, the only thing that is being calculated is continuous inference of the DNN for a few minutes with a constant random tensor of data. Thus,

30

the computing device will be maximally utilized with the DNN inference only. We consider it an upper bound of the power consumption and heat generation. The benchmarking reports increase in temperature and average timing of DNN inference. We additionally use Qualcomm Snapdragon Profiler software. It allows to capture hardware metrics of a running mobile application. From that, we visualize occupation of computing devices over time, and reason about potential bottlenecks.

**3.2 Mobile application development**

The Android mobile application was developed from scratch specifically for this Master the sis. A considerable amount of technical work was spent on implementing the general functionality of the application. This includes:

• system events handling, such as opening, closing, resuming from the operating system; • requesting of system permissions on camera and hard drive storage;

• adding binary files into an installation package, including runtime libraries, files of quantized DNN models, neural textures, configuration files, assets for inference of SMPL-X, etc; • setup of logging;

• reacting to user interaction, such as finger taps, swipes, pinch movements; • populating the user interface with options to select a particular DNN for inference, options of DNN input generation, predefined animations, features of rasterization, etc.

We publish only a part of the code base which is the most relevant to synthesis of avatar images on mobile, that is described in the following paragraphs (See Appendix A).

We initialize OpenGL library, by creating a Surface object from the Android SDK, that spans the whole screen of the device, and specifying it as a context of OpenGL to render to. This creates a separate execution thread on CPU. Only on this thread the commands of OpenGL (and thus GPU processing) can be executed. We configure the context to use 8 bits for all color channels of the screen and 32 bits for values in the Z-buffer. In OpenGL all the data is allocated on GPU as contiguous arrays, and the library has to be programmed how to interpret these arrays. The triangular meshes can be defined by a flat array of vertex attributes, where consecutive elements contain attributes of a single vertex. The most basic scenario is to pass an array of *⟨x, y, z, u, v⟩* with *xyz* being a 3D vertex position in a local coordinate space of the modeled object, and *uv* being texture coordinates of the vertex. Also, a connectivity array has to be provided in a form of indices in the array of vertex attributes. Three consecutive indices form a single triangle. We use back-face culling throughout the rendering, meaning that OpenGL will not render triangles facing away from the camera, slightly improving performance by culling some triangles out of

31

processing. To define which triangles are facing away, we specify to interpret vertex indices as clockwise triangles, meaning that a triangle of indices *⟨a, b, c⟩* will face towards the camera if and only if after projecting on the screen the vertices appear in a clockwise order. Overall, we use three separate triangular meshes, one for SMPL-X human body mesh (described further), and two ”meshes” which are essentially rectangles. The first one is rasterized, spanning the whole screen, and taking color values from a texture that contains a current camera frame. The other one is similarly rasterized, but at a location of the avatar, taking values from a texture loaded with output of the inferred DNN (See a debugging illustration from the mobile application on Figure B.17c).

OpenGL doesn’t support storage or rasterization of images with more than 4 channels (Red Green-Blue-Alpha only). We need to load the neural texture of *C* channels on GPU. If it is a multi scale texture, we convert it into a single scale, by averaging all the scales into the *C ×* 512 *×* 512 texture scale. Then we split it by 4 channels and load these parts as separate arrays on GPU. However querying of multiple textures requires extra processing time in OpenGL fragment shader programs. We thus create an OpenGL’s Array Texture data structure, that improves querying of multiple textures with the same texture coordinates, which is exactly how we use these texture parts. We use FP16 format for storage, to a reduce memory footprint of the texture compared to FP32. This also should result in better utilization of texture memory caches inside GPU. OpenGL allows to apply an additional filtering algorithm on textures, to generate multiple scales of the texture, which are called MIP-maps (see Figure B.6a). Those can be used to render triangles that span very little area on the screen. The filtering prevents blurriness and aliasing artifacts from appearing due to bilinear sampling. Furthermore, anisotropic filtering can be applied to also generate stratched variants of scales. It improves quality of rendering triangles that face at steep angles to the camera (see Figures B.6b and B.7). Both methods require extra texture memory: about x2 for MIP map ping, and up to x16 for anisotropic filtering, and slightly more processing time. However, given the small resolution of our textures and relatively low triangle count in the body model, we found the performance difference to be negligible. We would like to clarify the confusing terminology. The multi-scale texture exists only in the DNN pipeline in Python, the scales of it are upsampled and averaged before rasterization. The texture MIP maps in the mobile application always remain at lower resolutions, and a GPU driver heuristically selects an appropriate MIP-map to render a certain triangle.

Next we initialize a tracking session of ARCore library. It starts intercepting and processing the camera frames on GPU, allowing to refer to the camera data as an OpenGL texture. By default, ARCore processes a new camera frame and awaits for the next to arrive. The application is then capped to the camera’s update rate – 30 frames-per-second (FPS) for our devices. We configure

32

ARCore to never wait and process the current camera frame, regardless if it was already processed. This allowed us to cap the application to the Android’s internal limit of 60 FPS.

As was mentioned in Section 3.1, we initialize SNPE DNN models with ”Sustained High Performance” profile, and the computing unit specified by the user (supporting either GPU or DSP). GPU offers better support of neural layers used in DNN models, since it can perform computations in FP32 or FP16 format, analogously to a desktop GPU. But running a DNN on it would also mean that input rasterization and DNN inference would have to be done one after another, possibly bottlenecking the performance. On the other hand, DSP offers much higher performance than GPU, and inference can take place independently and in parallel with our input generation on GPU. However, the usability of DSP is very limited, e.g. much fewer neural layers support quantized computations; also loading of the DNN models requires long initialization, up to several minutes. Besides, Qualcomm Snapdragon’s DSP contains a hardware security policy, that only allows to run on consumer devices the programs, that were certified by the hardware vendor. We ignore this issue for our project, by having test engineering samples of smartphones that suppress this security check. If SNPE fails to load a model on DSP, it may fallback to using GPU, or even CPU execution, obviously being extremely slow for inference.

Finally, before inference, we can pre-compute parts of the SMPL-X [74] body mesh. Since it’s only done once, we do this computation on CPU. We load 3D positions of *N* vertices in the template mesh with a rest body pose. Then, following the algorithm (see at the end of Section 2.2), for mobile inference we’ll use the same body shape vector *β* that was used during DNN training. From *β* we can compute a linear component *BS*(*β*; *S*) as in Formula 2.8, that offsets the template vertex positions to represent the specific body constitution of the avatar. We compute it is as a multiplication of a tensor with shape *N ×* 3 *× |β|* (provided by SMPL-X) and the *β* vector (from the dataset), yielding the offsets *N ×* 3 that we add to the template vertex positions. We similarly compute the *BE*(*ψ*; *E*) linear component, that offsets vertices of the face. Although the face shape can slightly change between different facial expressions, we only compute the linear component once before inference, for simplicity. Lastly, SMPL-X includes a pretrained linear regressor of joints 3D positions, from all the mesh vertices. We also pre-compute this regression, which comes down to computing multiplication of the regressor’s matrix 55 *× N* and the mesh vertex positions *N ×* 3, yielding the 3D positions of 55 body joints.

During inference, the pose-dependent linear component has to be computed and added for each frame in order to get a posed body mesh. As was already described, it involves finding a weighted sum of 55 affine transformation matrices of size 4 *×* 4, with a separate set of weights for each vertex. The weights define how strong the rotation of a particular joint affects offset of the vertex (for example, see Figure 3.4, how some vertices are dominantly affected by a particular

33

nearby joint). Having approximately 10 000 vertices in the mesh, we need to multiply and add a total of 55*·*10000*·*4*·*4 = 8800000 floating-point values on every frame. This computation would be time consuming on CPU, but since it can be trivially parallelized, we instead pre-allocate the array of weights in OpenGL (on GPU), and compute the weighted sums in the vertex shader program. We further noticed, that among 55 weights per vertex at least 43 are zeros. Thus we can improve the performance, by sorting the weights on CPU during the pre-processing step, and allocating on GPU only 12 most significant weights and their joint indices *i ∈ {*1*, ...,* 55*}*. Overall, we load on GPU the pre-computed shaped vertex positions *⟨x, y, z⟩*, texture coordinates *⟨u, v⟩*, joints weights *wk* and joints indices *ik* as an array of vertex attributes on GPU. We use an interleaved layout, where *⟨x, y, z, u, v, w*1*, .., w*12*, i*1*, .., i*12*⟩* of a single vertex are located contiguously in the GPU array, improving memory cache efficiency of posing a single vertex.

Next, we will describe the interactive stage of the application, which is an infinite loop of computing and rendering the AR experience. Each frame starts by inferring the pose-dependent linear component *BP* (*θ*;*P*) of SMPL-X. It is computed from a pose vector *θ*, containing 3 *·* 55 values with axis-aligned rotation vectors for all the joints. A single rotation vector **r** = *⟨rx, ry, rz⟩* needs to be converted into a 3 *×* 3 rotation transformation matrix **R**, using Rodrigues [77] formula

(we use **I** for an identity matrix):

**R** = **I**3*×*3 + sin(*∥***r***∥*) *·* **A** + (1 *−* cos(*∥***r***∥*)) *·* **A**2*,* where **A** =1*∥***r***∥*



0 *−rz ry*

*rz* 0 *−rx −ry rx* 0



(3.2)

The joints apply their rotations and translations recursively, e.g. a position of a finger can be found from sequentially applying transforms of the arm, forearm, palm, and every knuckle. We could compute each joint’s final transformation independently, by multiplying all chains of the matrices. However, we can exploit the fact, that the joints form a tree structure, i.e. every joint depends on only one parent joint without loops. Thus, we use a topologically ordered array of 55 joints, meaning that no joint in the array precedes its parent joint. This allows faster CPU computation for 4*×*4 transformation matrices of joints, because by the time we process a joint *i*, all its dependent transformations are already computed in a form of a single matrix of its direct parent. We take 3D locations of joints **t** (regressed during the initialization stage), and get relative joint translations ∆**t** by subtracting joints positions from their parents’. The final recursive transform **H**

for a joint *i* is found as:

**H***i* = **H***p ·*



**R***i*3*×*3 ∆**t***i*3*×*1 **0**1*×*3 1



 *·*



**I**3*×*3 *−***t***i* **0**1*×*3 1



 (3.3)

where *p* is a parent index, the transformation matrix **H***p* of which is already found by the time we compute *Hi*. The first matrix multiplication computes a relative recursive transform for the joint, and the second multiplication subtracts the true posed location of the joint. From the programming

34

point of view, to save on computations, we implement the matrix multiplication operations not naively, but by expanding and merging the operations together, e.g. computing each element of **R** matrices in place. We try to iterate the data in memory in a cache-friendly way, to write the data consecutively. We expect that it helps to retain exclusive usage of cache lines, without overhead of cache synchronization. For the full SMPL-X inference implementation, head to Listing A.6.

Then, we extract tracking information from the ARCore session, namely camera intrinsic and extrinsic matrices, as well as transformation matrices of the tracked anchors. We pre-multiply the anchor’s matrix by camera extrinsic matrix, and then by the intrinsic matrix, obtaining a single transformation from local body mesh coordinates to projected image coordinates. Unlike the classic representation of intrinsics, as in Formula 2.4, OpenGL uses an equivalent representation, but with another notation, called Native Device Coordinates. Formerly, the projection frustum was defined with a 3*×*4 matrix, parameterized with image width and height *W*/*H*, focal lengths *fx*/*fy* from the pin-hole to the image, and image center offset *cx*/*cy* in pixel coordinates. In OpenGL, the frustum is set by a 4 *×* 4 matrix via 6 distances that bound the frustum: top *t*, bottom *b*, left *l* and right *r* planes; a near plane *n*, analogous to focal distance; and a far plane *f*, which defines the maximum depth that is mapped by the virtual camera. See Figure 3.1 illustrating the meaning of the frustum parameters. The matrix has 4*×*4 size in order to additionally compute depth of each pixel, allowing to store it in a Z-buffer, and querying if some rasterized pixels are being occluded. We can represent

the same projection with either notation:









2*fx*

*W*0*W−*2*cx*

*W*0



OpenGL

2*n*

*r−l*0*r*+*l*

*r−l*0



(3.4)

**KNDC**

previous notation =

02*fx H*

*H−*2*cy*

*H*0

notation =

02*n t−b*

*t*+*b*

*t−b*0

0 0 *−f*+*n f−n*

*−*2*fn f−n*

0 0 *−f*+*n f−n*

*−*2*fn f−n*

0 0 *−*1 0

0 0 *−*1 0

There are a few changes required for this projection matrix. Since we use it to rasterize an input frame to the DNN, we have to supply square images (512 *×* 512 pixels resolution). However the default projection matrix has aspect ratio of the device’s screen. The frustum can be squeezed, e.g. along vertical *y* axis, by multiplying the second row of **KNDC** by the screen dimensions ratio *W*/*H*. Additionally OpenGL stores images from a bottom row to top, meaning that it appears vertically flipped when we use it outside OpenGL. To solve it, the element at second row and second column can be negated.

Now that we have posed joints, in order to pose all the mesh vertices we send the trans formation matrices **H***i*, as well as the merged matrix of extrinsics and intrinsics on GPU, as uni form1 variables of the vertex shader program (see Listing A.7). Here, for each body mesh vertex we combine the most significant joints weights with corresponding matrices, taken by index off

1The term *uniform* is specific to OpenGL, it means that a single variable value is shared across all the primitives (vertices/pixels) processed by a shader.

35

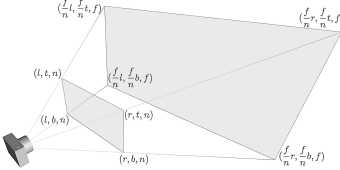


Figure 3.1: A frustum of the pin-hole virtual camera model. It is a pyramid, where apex is both the camera position and the origin of coordinates. *x* and *y* axes align with the camera’s right and top directions respectively, *z* axis faces straight in the front (in some notation straight behind). An image is virtually placed at a *z* = *n* coordinate (referred to as near plane, or focal distance), parallel to the *xy* plane. The frustum’s extents along *x* (*y*) axis are specified with left *l* and right *r* (bottom *b* and top *t*) parameters. An image is formed by projecting the frustum’s content on the near image plane. It is common to limit the frustum with a far plane *f*, so that a finite-bit representation can be used in a computer to store depth information used for occlusion testing. Image from kor.faithcov.org/749684-gl-modelview-and-gl-projection-WBGZIU

set specified in the vertex attributes. Notice, that when we compute uniqueVertexTransform matrix, we don’t add up the matrices plainly like *a* + *b* + *...* + *g* + *h*, instead we use a scheme ((*a* + *b*) + (*c* + *d*)) + ((*e* + *f*) + (*g* + *h*)). Due to the order of operations execution, the former variant requires to compute *a* + *b*, and only after that compute (*a* + *b*) + *c* and further, even if hardware has multiple Multiply-Accumulate circuits (MAC) available. With the latter variant, we force computation of 4 inner sums independently, and then add them into the final matrix. This little reordering can optimize the performance, since its effect scales with the number of vertices. The vertex shader terminates by transforming the vertex with the affine transform, and projecting it into the screen coordinates.

Then OpenGL automatically performs rasterization into an image. OpenGL interpolates ver tex attributes and passes them into a fragment shader program, which is executed for all image pixels (see Listing A.8). We use interpolation of texture coordinates, to sample the neural texture in agreement with the texture unwrapping, and construct the pixels of the DNN’s input tensor. SNPE processes the tensors in Batch-Height-Width-Channel layout (BHWC), i.e. all channel val ues of a single pixel are followed by channel values of the next pixel. On the bright sight, OpenGL uses exactly the same layout for rendered images. However once again, we stumbled upon a limita tion of OpenGL that it cannot generate images of more than 4 channels, while we need *C* channels. One solution is to render *⌈C*4*⌉* images of size 512 *×* 512 *×* 4 using OpenGL on GPU, and then to permute the data in CPU memory to make a single tensor 512 *×* 512 *× C*, and pass it to the DNN.

36

However, it was proven to be extremely slow, even if this permutation is embedded into the DNN architecture, which may get some acceleration from the DSP device. Instead, recall that SNPE works with quantized input data, with INT8 per each channel value. We found a way to generate in OpenGL images with 4 bytes allocated per each image channel, thus having up to 16 bytes per pixel. Now, by passing quantization constants into the fragment shader, we can sample the floating point numbers from the texture, and using the quantizing transformation (see Formula 2.17) we obtain an INT8 value, which can be bit-shifted into an arbitrary byte of a particular channel. Thus, we can successfully pack input images with up to *C* = 16 channels using a single rasterization, without requiring *any* additional processing on the DNN side - it can immediately start inferring from this data layout. The background pixels on the image that don’t belong to the body mesh have to be handled carefully. By default OpenGL assigns 0 to them, but with the quantized com putations, an integer 0 is actually mapped to the minimal value in the input tensor. Thus, using quantization constants we configure OpenGL to fill these pixels with an integer value, that maps to a floating point 0, exactly as the neural network expects.

After generating the input frame, it has to be queried on CPU side from the OpenGL. As a destination of this frame, we use the pre-allocated buffer that was mapped to input of the SNPE DNN instance during initialization stage. A technical detail, is that OpenGL may not efficiently work when querying the image data and rendering to it on the same frame. It can attempt to allocate additional GPU memory and render the next image in here, while the current image is being read. To reduce the latency, an *orphaning* approach is used, where after rendering we issue a command to free the image memory, and then we manually request to allocate new memory for the image. Non-intuitively, OpenGL may instead reuse the memory that was just freed, without allocating new memory on its own. We also tried to use an asynchronous mechanism of querying OpenGL images, i.e. we send a request to get the data, but we do not block the execution and process other tasks while the data is being prepared (in our case, we spend time compute SMPL-X components for the next frame). Unfortunately, the approach was slower than the default sequential querying with orphaning, perhaps due to a limited hardware support on our platform.

Finally, we issue the command to infer the DNN from the given input data, and draw all the AR content on the screen - the camera frame, the tracked surfaces planes, and the synthesized avatar image, which we have to pass to OpenGL as a texture. Besides rendering the content in AR, we also support infering the DNN on predefined camera and SMPL-X poses. We dump the resulting images onto the hard disk of the device, and later compare the visual quality of inference on desktop hardware, on mobile GPU or DSP.

This concludes the minimal necessary actions for the real-time neural rendering of avatar im ages in AR. We then notice, that the visual quality of avatars can drastically suffer when observing

37

the avatar from far away. The neural network is unlikely to be trained to see all the miniature scales of the inputs, thus it may output unpredictable results, possibly failing in segmentation (Figure B.17a). Moreover, the default stretched projection appears pixelated, primarily because the resolu tion of output images (512 *×* 512) is a lot smaller than of the screen (on our devices 1920 *×* 1080). Thus, when we render the output avatar image, it gets noticeably blurry. This effect is amplified, when looking at the avatar from close up (Figure B.18a). Besides being low resolution, a big part of the frame will be never seen on the screen due to the extra projection stretching we performed to rasterize images as squares (Figure B.10a), however computation time is spent on it every frame.

The solution that we refer to as *dynamic cropping*, helps to render a single avatar with higher quality. For each frame of avatar rendering, we can take posed joint positions **t** as in Formula 3.3, project them into image coordinates using multiplication by the extrinsic matrix and the modified projection matrix. Then by finding the minimum and maximum *x* and *y* image coordinates we obtain an image-space bounding box, that would encase all the extents of the person (Figure B.10b). For simplicity, we compute the bounding box aligned with *x*-*y* axes on the image, although a rotated bounding box can yield better results, with more complicated implementation as well. If we were to rasterize the input image only within this bounding box, then the input avatar would always appear in a full-body (FB) scale, for which the DNN is trained to output the highest quality. Still, a smaller part of the frame can be computed in vain, in instance when the avatar is not observed in full-body (Figure B.10b). Going further, we can reduce the wasted frame parts to the minimum, by clipping the bounding box to the screen boundaries, thus supplying the DNN with a slightly zoomed image of the avatar (Figure B.10c). Overall, this allows to extract maximum resolution from just 512 *×* 512 generated frames, they can actually appear quite sharp and detailed from both far-away and close-up views.

Now we will describe how cropping of the projection frustum to the bounding box works mathematically. We have a default projection matrix queried from ARCore library, that spans the whole screen of the device (see Formula 3.4). Among the frustum parameters we can arbitrarily select near and far plane distances (traditionally taken *n* = 0*.*01 and *f* = 100*.*0). We are not given the frustum parameters top *t*, bottom *b*, left *l*, right *r*, however by taking an arbitrary point in homogeneous camera coordinate space *⟨x, y, z,* 1*⟩*, and projecting it using the multiplication by

projection matrix, we can get image-space projected coordinates:

*ximg* = **KNDC**1*,*1*· x* + **KNDC**1*,*3*· z* =2*n*

*r − lx* +*r* + *l*

*r − lz* (3.5)

*yimg* = **KNDC**2*,*2*· x* + **KNDC**2*,*3*· z* =2*n*

*t − bx* +*t* + *b*

*t − bz* (3.6)

*f − nz* +*−*2*fn*

*zimg* = **KNDC**3*,*3*z* + **KNDC**3*,*4 = *−f* + *n* 38

*f − n*(3.7)

However, the projected Z-coordinate of the point *zimg* is known - it is the distance from the pin-hole to the image, i.e. the near plane distance *n*. Also see Figure 3.2, where *y* (or *z*) would be a distance between *y* = 0 level (*z* = 0 level) and the green dot on the person’s head. Then *yimg* (*zimg*) would be a distance between *y* = 0 level (*z* = 0 level) and the corresponding green dot on the near plane. Using similar triangles, we can get relations *yimg*

*n* =*yz*and *ximg*

*n* =*xz*. We cannot

solve the Equations 3.5-3.7 for parameters *t*, *b*, *l*, *r*, *n* and *f*. However, we can represent them in terms of elements of **KNDC** matrix, thus obtaining a bijective mapping between the projection

matrix and the frustum parameters:

*n* =**KNDC**3*,*4

**KNDC**3*,*3 *−* 1*f* =**KNDC**3*,*4

**KNDC**3*,*3 + 1(3.8)

*l* = *n ·***KNDC**1*,*3 *−* 1

**KNDC**1*,*1*r* = *n ·***KNDC**1*,*3 + 1

**KNDC**1*,*1(3.9)

*b* = *n ·***KNDC**2*,*3 *−* 1

**KNDC**2*,*2*t* = *n ·***KNDC**2*,*3 + 1

**KNDC**2*,*2(3.10)

Figure 3.2: A 3D space with origin at the camera, and axes aligned with camera directions (*x* - right, *y* - top, *z* - front). The figure shows a slice of this space with *x* = 0, illustrating the dynamic projection crop. *(Left)* The near plane of the old screen frustum is defined by a *z* distance (arbitrary), and *x*/*y* coordinates of the frustum borders (top/bottom coordinates on illustration). *(Right)* To make the projection cropped around the avatar body, we compute an axis-aligned bounding box of the body, then project its spans onto the near plane, obtaining top/bottom borders, defined by intersection coordinates of the projection lines and the old near plane. The new frustum will have the same near and far planes as the old one, to ensure preservation of perspective.

Now we need to compute the new cropped frustum parameters: top *tnew*, bottom *bnew*, right *rnew* and left *lnew* (see Listing A.4). We can take *nnew* = *n* and *fnew* = *f*, since they are unaffected by the cropping operation. We take 3D locations of avatar joints for the given frame, and from minimum and maximum joint coordinates we compute the crop’s center *⟨x, y, z⟩* and crop extents *⟨H, W⟩* in the camera coordinate space. Then, values *Hz*and *Wz*tell how many times the crop area is smaller relative to the original frustum. We can multiply *t* and *b* by *Hz*; and *l* and *r* by *Wz*respec

39

tively to shrink the frustum area to the desired size. Now the frustum center needs to be shifted to match the crop in the camera space coordinates. For this we multiply the crop center’s coordinates *⟨x, y⟩* by *nz*thus getting projected coordinates of the crop center, and by these coordinates we offset previously scaled *t*, *b*, *l* and *r*. Again, see how the old and new frustums correspond to each other on Figure 3.2. Overall, the computation of the cropped frustum looks like:

*nnew* = *n fnew* = *f* (3.11) *lnew* = *l ·Wz− x ·nzrnew* = *r ·Wz*+ *x ·nz*(3.12) *bnew* = *b ·Wz− y ·nztnew* = *t ·Wz*+ *y ·nz*(3.13)

Using Formula 3.4, we make a new projection matrix from the updated frustum parameters. The matrix can be passed to the previously described GPU shader programs, to render the images. However due to cropping, the inferred avatar image has to be rendered at a specific image location, that corresponds to the initial cropping bounding box in the image space. Since we use a sepa rate rectangle mesh for rendering the inferred images on top, before using the dynamic crop the coordinates of the rectangle corners were:

top\_left = *⟨−*1*,* 1*⟩* top\_right = *⟨*1*,* 1*⟩* (3.14) bottom\_left = *⟨−*1*, −*1*⟩* bottom\_right = *⟨*1*, −*1*⟩* (3.15)

We need to map these coordinates to the cropped frustum’s parameters. Since the old frus tum’s parameters do not necessarily form ranges between -1 and 1, we need to take a corresponding parameter of the cropped frustum and find how it relatively adjusts the old one. For example, a value of relative update by the *rnew* parameter would be *rnew*

*l−r*. Thus, the updated image coordinates

to render the dynamically cropped image would be:

*l − r,tnew*

*t − b⟩* top\_right = *⟨rnew*

top\_left = *⟨lnew*

*l − r,bnew*

*l − r,tnew*

*t − b⟩* (3.16)

bottom\_left = *⟨lnew*

*t − b⟩* bottom\_right = *⟨rnew*

*l − r,bnew*

*t − b⟩* (3.17)

**3.3 Improving quality of synthesized images**

The baseline DNN architecture [36] at the moment of publication was state-of-the-art in syn thesizing full-body (FB) human avatar images, after learning from a few images (or a video se quence). However, it may not be enough for the AR scenario in a mobile application. In instance, in AR it is possible to observe the avatar from far-away, close-up from top and bottom, etc. The baseline DNN is not trained to generate realistic images for arbitrary views on the avatar. The ar chitecture definitely cannot extrapolate its knowledge to synthesize at least meaningful images on those out-of-training scales. The reason lies in the convolutional nature of this architecture. Con

40

volutional filters are optimized to produce the biggest activation values when encountering certain patterns on an image. The zoomed in (or out) images have the same patterns globally, but not locally within the receptive field of convolutions. See Figure B.23, where the baseline is trained only on FB inputs, and it cannot generate as good images for the upper-body or close-up face.

Also, the DNN is trained on images with a single person and only slight variation of their appearance. The target for the neural renderer is to fit those training images to reproduce the similar level of realism, as the original photos. On the other hand, this fitting may be uncontrollable, resulting in a perfect reproduction of the training samples, but failing to generalize the synthesis of novel camera views or body poses. Consequently, since the architecture relies a lot on fits of the SMPL-X body model, the DNN tends to overfit for any slight misalignment of the mesh with ground truth images. As a result, even slight adjustment to a view angle or pose may lead to drastically different synthesized images. We refer to the visual artifacts that appear from such overfitting as *spurious correlations*. When DNN processes a body part posed in a way, which is similar to some training sample, the rendering of the whole avatar may change, even for totally unrelated body parts or very different poses (see Figure B.13).

The issue of rendering far-away images of the avatar can be solved in the AR application, using the dynamic projection cropping as we already described in Section 3.2. However, there is no other way to generate close-up images but to train the network to do so. The current experimental setup for the training allows to do affine image-space augmentations, and scaling the images in particular. Multiple image types need to align after augmentation: an input frame, a ground truth image, a segmentation mask. To keep the alignment, it would require to first rasterize the body mesh with a neural texture in a full-body scale, to align the rasterization with the ground truth image, and then to apply the same affine image-space transformation to both. Such transformation is equivalent to multiplying a grid of pixel coordinates by an affine transformation matrix with size 3*×*3 (like a 3D extrinsic in Formula 2.5, but 2D). Then, for each pixel of a resulting image, bilinear interpolation is computed between 4 neighboring positions in the transformed coordinate grid.

Although it would be sufficient to make zoomed-out augmentation for the images, the whole approach has a flaw for making zoomed-in images. The body mesh triangles appear very small on full-body scale images, frequently as a single pixel. However, in the neural texture the triangles may be encoded with many pixels. After rasterizing and applying a strong image-space zoom ing, the fine-grained details from the neural texture will not be added. Instead, the low-resolution values will be interpolated, and thus averaged. Consequently, the parts of the neural texture will be optimized with averaged gradients, instead of pixel-specific ones. This prevents learning of sharp patterns from the ground truth in the neural texture, although intuitively it could improve the neural rendering. Besides, with this approach, the visual quality of the ground truth images will

41

decrease twice from bilinear interpolation, the first time when aligning with the input rasterization, the second time when applying the image-space augmentation.

Since we control the process of the input frame rasterization, instead of applying image space augmentations, we could modify camera matrices (see Formulas 2.4, 2.5) that are used for rasterization. With that, the rasterization will not need any bilinear interpolation, and can be made arbitrarily sharp by rasterizing in a higher resolution. The ground truth images still require image space augmentation, but we no longer need the aligning operation, since we do not augment the input frame in the image-space. Thus, at most one bilinear interpolation will be applied, preserving a little bit more of quality and realism. Moreover, we can augment the ground truth in the image space in full original resolution, rather than the input frame’s resolution (which is fairly low – 512 *×* 512 pixels). We implement this approach as a Python module, and call it *camera-space augmentations* (see Figure 3.3).

(a) (b)

Figure 3.3: (a) Implementation of zooming using image-space augmentations. The body mesh is rasterized in full-body, the ground truth is transformed once to be in an alignment with the rasteriza tion. Then both are magnified, loosing a lot of quality. (b) Using camera-space augmentations, the virtual camera used for rasterization replicates zooming by moving towards the avatar. The input image can be made arbitrarily sharp. The ground truth image is still transformed with image-space augmentations, but only once, preserving a bit of quality.

The module allows specifying pixel coordinates of a point on the ground truth image, which will be a center of rotation. A default center of rotation is in the middle of the image (with fractional coordinates, if dimensions are odd). By applying a random image-space augmentation using Kornia library, we extract specific values of rotation *θ* angle (in radians), scale multipliers*⟨sx, sy⟩*(negative values mean additional horizontal or vertical flipping) and translation *⟨tx, ty⟩* (in pixels) that were applied to the ground truth. We then construct 4 *×* 4 rotation **R** and scale **S** matrices (see Formula 3.19), and by combining them as **E** = **R** *·* **S**, we use it as an extrinsic matrix for rasterization of the body mesh. Note, that there is no rotation around *z* axis, which would be wrong to apply as it changes how the image is perspectively projected during rasterization.

42

**R** =



*cos*(*θ*) *−sin*(*θ*) 0 0

*sin*(*θ*) *cos*(*θ*) 0 0 0 0 1 0 0 0 0 1



=**R***xy* **0**2*×*2 **0**2*×*2 **I**2*×*2



 (3.18)

**S** =



*sx* 0 0 0

0 *sy* 0 0 0 0 1 0 0 0 0 1



=**S***xy* **0**2*×*2 **0**2*×*2 **I**2*×*2



 (3.19)

The camera translation is harder to apply using the extrinsic matrix, because we need to rasterize the input frame in a resolution (512 *×* 512), which is different from the one the image space augmentation was applied (1920 *×* 1080 pixels of the ground truth images). Thus, we apply it by changing the center of projection in the intrinsic matrix, i.e. *cx*, *cy* as per Formula 2.4 (updated values are denoted with *c*ˆ*x*, *c*ˆ*y*). The translation augmentation is traditionally applied after rotation and scaling, thus the projection center is also transformed by **R***xy ·***S***xy*. We then offset the projection center by the translation values from the ground truth augmentation, and do it relatively to the coordinates of the augmentation center*⟨dx, dy⟩*. We find a ratio *z* of the ground truth and destination resolutions (for which the DNN is trained). We finally multiply the offset by this ratio to correctly align with the ground truth.



*c*ˆ*x c*ˆ*y*



 =



*cx cy*



 + *z ·*



**R***xy***S***xy ·*



*cx cy*



 +



*tx ty*



 *−*



*dx dy*

 



 (3.20)

Further, we will describe the experiments on DNN training, performed on top of applying camera-space augmentations to train on zoomed images. We will group experiments with similar ideas into paragraphs, which we summarize with a bold emphasis on the font. We also put ref erences to figures that contain such experiments, although the discussion of them is provided in Section 4.2. We explain the motivation and implementation details where needed. For brevity, we present paragraphs and experiments within them not necessarily in the same order, in which we conducted the research.

**Body joints as augmentation centers** (Fig. B.24, B.25)

Having the Python module for computing camera-space augmentations from the image-space transform, we need to define to which points on the ground truth image we should zoom. Obviously, choosing a random location on the image would not work, as the ground truth frames contain background content. If a point outside the human body is chosen, given that the zooming scale is strong, the person will not be present on the augmented image at all. Consequently, the input

43

rasterization for the DNN will be an empty image. We decided for each frame in the training sequence to use predicted SMPL-X joints locations as augmentation centers. Since they are in 3D, we project them onto the image using a projection matrix that was used for SMPL-X fitting process. Then, we take a subset of the joints, for each we assign a specific zooming scale range and probability of being chosen as an augmentation center. Finally, during training we sample random joints for a batch of images according to the weights, uniformly sample a zoom scale from the corresponding range of scales, and apply the camera-space augmentations to the projected locations of joints. We perform experiments, where we select a subset of major joints: head, feet, palms, pelvis. We experiment with different maximum scale of the zooming range: 2.0, 3.5, 4.5, 8.0. Since only a subset of the joints is used, it is possible that other intermediate body parts may be observed slightly less frequently.

**Body mesh vertices as augmentation centers** (Fig. B.26)

Choosing body joints as augmentation centers is easy to apply. However we suspected, that both the neural renderer and discriminator might overfit to certain body parts appearing more fre quently at the certain parts of the screen. At the very least, the joint locations are more biased to be in the center of the frame. We tried to increase the variance in the zooming augmentation, by instead choosing body mesh vertices as augmentation centers. We also wanted to preserve the ability to choose specific weights to certain body parts and their zoom ranges. Thus, we decided to build an association between joints and mesh vertices. We used weights of the joints regressing model from SMPL-X. We noticed, that a certain joint is regressed with the highest weights from the nearby vertices. Hence, we associate a vertex to the joint with the highest regressing weight (see Figure 3.4 with examples for a few joints). We also need for each ground truth image to find 2D projections of all the vertices, similarly to the joints projections. Then, to compute camera-space augmentations we would randomly sample a joint, then uniformly sample a random vertex attached to the joint, and uniformly sample a scale of zooming associated to the joint.

We perform experiments with the same distribution of joints and zoom ranges as in the ex periments above. We also experiment with making a fair distribution of zooming to all body parts. Note that taking the same weights of joints will not accomplish that. This is because the SMPL-X body mesh has different density of vertices, there are about as many vertices in the head as in the other body parts combined. Thus, we tried to take all vertices associated with a joint, find their extents along *x*, *y* and *z* axes, and compute a volume of this bounding box. Then by taking a ratio of the number of joint-associated vertices to the volume, we find the density of vertices around a single joint. For camera-space augmentations, we try to take joints weights, that would equalize this density. Thus we hope to achieve the uniform distribution of body parts images during training.

44

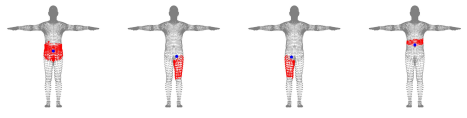


Figure 3.4: Assignment of mesh vertices (red) to body joints (blue). For each vertex, we take the joint with the most significant skinning weight from SMPL-X body model, that determines how movement of the joint affects the given vertex’s movement. We use this assignment to select mesh vertices as centers of camera-space augmentations. We speculate that choosing vertices rather than plain joints improves augmentations variance and decreases overfitting.

**Increased learning rate for the neural texture** (Fig. B.27)

During experiments with zooming, we observed that gradient tensors of the neural texture have norm values that are typically an order of magnitude lower than of the neural renderer or discriminator. We experimented with increasing the learning rate of the neural texture by either 5 or 10, to compare the outcomes.

**Train with zoomed scale and fine-tune with FB scale only** (Fig. B.28, B.29)

From the above experiments we noted, that by training on zoomed images, although we indeed achieve more realistic results on zooms, the quality of FB images decreases, compared to the baseline. On one hand it is logical, because the neural renderer is much less frequently tasked with rendering FB scale images. On the other hand, by training on FB scale only, the ground truth images in resolution 512 *×* 512 lack a lot of fine-grained details, partically due to bilinear interpolation in the augmentations. We expected that by seeing zoomed images, the neural renderer and the neural texture will learn more of such details from the ground truth images, possibly improving the FB scale as well. However, we did not observe FB improvement. We then tried to train the architecture on zoomed images with further fine-tuning for FB images.

We tried to decrease learning rate for the neural renderer and the neural texture by some factor. We try the following combinations of factors (the 1st for the neural renderer, the 2nd for the neural texture): (1*,* 1), (0*.*5*,* 1), (0*.*2*,* 1), (0*.*1*,* 1), (0*.*5*,* 1), (0*.*5*,* 0*.*5), (0*.*2*,* 0*.*5), (0*.*1*,* 0*.*5).

**Collecting BN statistics only on either FB or zoomed scale** (Fig. B.30, B.45, B.31, B.47)

We had a suspicion, that BN layers inside the architecture may behave wrong, when normal izing batches of data with different scales of zooms. For example, when the neural renderer or discriminator processes a frame with a close-up on either face or foot, the statistics of the batch may have high variance, and averaging them may result in gradual instability of the training. On

45

the other hand, FB images are more consistent, they always show both the head and feet in the similar proportions. Thus, we separately pass batches of FB scale and zoomed scale images. We use batches of FB 20% of times, and batches of zoomed frames 80% of times. We experiment with disabling BN layers across the whole architecture, either only for zoomed batches, or conversely only for FB batches (as a wild hypothesis).

Note that in PyTorch terms, disabling of BN means that the affine parameters will not be optimized, and the running average statistics will not be updated, and they will be used for nor malization of the data. Additionally, we experiment with using batch statistics instead of running average statistics to normalize the data when BN layers are disabled. We also try passing zoomed images much more rarely, as well as combining it with zooming to body mesh vertices uniformly.

**Alternative normalization layers instead of BN** (Fig. B.32)

We continued to analyze the aforementioned discrepancy of batch statistics when training on zoomed scale. We try to replace the BN layers with other normalization layers across the whole architecture. We try Instance Normalization layers [96], Group Normalization layers [103], as well as removing all the BN layers. Since we train both the neural renderer and discriminator from scratch, there are no architectural issues with replacing BN layers inside the well known architectures that are used in the baseline (ResNet18, StyleGAN’s discriminator).

**Instance Normalization layers instead of BN** (Fig. B.33, B.34)

After noticing, that Instance Normalization layers both perform better and can be launched on our mobile hardware, we check that the visual quality is admissible when using other train ing sequences. Since in PyTorch by default Instance Normalization layers do not collect running instance statistics (as BN layers do) that are used for inference, we also try to enable this behavior.

**Removing BN layers from parts of the architecture** (Fig. B.35, B.36)

In pursue of finding which BN layers in the architecture are prone to fail more, we try to partially remove BN layers: from the neural renderer; from the last layers of the neural renderer; from the discriminator; from both the neural renderer and discriminator; from the neural renderer, and replacing discriminator BN layers with Instance Normalization layers. By doing so, we expect that the visual quality could improve for some experiments, which would imply the correctness of our hypothesis on batch statistics of zoomed images. If otherwise the quality decreases, then the issues might be anywhere else, or prove that normalization layers of any kind are crucial for convergence, even at a cost of visual artifacts or overfitting.

46

**Using BN layers without learned affine parameters** (Fig. B.37, B.38, B.39)

Apart from the running average statistics, there are also learned affine parameters in the Batch and Instance Normalization layers (*γ*, *β* as in Formula 2.6), which also may fail to converge prop erly because of the zoomed scale images. We thus try to make these parameters constant *γ* = 1, *β* = 0. We experiment with applying it:

• in the neural renderer;

• in both the neural renderer and discriminator;

• with normalization statistics collected only on FB frames, and without affine parameters in the discriminator (or discriminator with all Instance Normalization layers); • with normalization statistics collected only on FB frames, and without affine parameters in the discriminator or both neural renderer and discriminator;

• with normalization statistics collected only on FB frames, all BN layers replaced with In stance Normalization layers, and without affine parameters in both neural renderer and dis criminator;

**Modifying optimization of neural texture’s unused parameters** (Fig. B.40, B.41, B.42, B.43)

Adam [54] optimizer that is used for all learnable parameters in the baseline architecture, stores momentum of all the parameters’ gradients in a form of exponential average. In theory, this helps to get stability of optimization when encountering abnormal or invalid training data. Making an optimization step with a gradient, which was resulted from such data may be harmful. The momentum prevents too abrupt adjustments to the previous direction of optimization.

When we rasterize input frames with a neural texture, only at most half of the texture’s pixels will be present. For example, when rasterizing a frontal view of a person, all the pixels for the back view will be absent. Thus, only the seen portion of the pixels will receive a non-zero gradient value. Although all the other pixels will not update on the current step, their momentum will decay in Adam optimizer. We hypothesize, that we can improve the training, by changing the Adam’s algorithm to prevent decaying of momentum for parameters that received no gradient on an optimization step. Our motivation, is that the momentum direction that is currently stored for the unseen pixels of the neural texture will still be fairly correct even after updating the seen pixels, because they store very independent information.

We apply this adjustment by reusing Adam algorithm implementation from PyTorch, and by using masked operations when decaying the momentum tensors. The mask filters out parameter positions, where the computed gradient values are near zero. We train with this modified optimizer and compare it with a normal train on zoomed images (to the joints). We then add other adjustments, such as collection of BN layers statistics only on FB scaled frames and then:

47

• choosing body mesh vertices as centers of augmentations;

• fine-tuning on FB scaled frames;

• trying a batch size equal to 1, so that batch statistics will effectively be instance statistics, possibly proving that discrepancy of batch statistics may cause the visual artifacts (see the experiment’s results on Figure B.44).

**Disabling GAN losses on zoomed scale** (Fig. B.45, B.46)

The learning from competition, which is what the GAN approach in essence is, may be both advantageous and harmful. This may apply to the zoom scaled images as well. If the discriminator network is too accurate in distinguishing the real zoomed images from synthesized ones, then the generator may never succeed fooling it. This example of a ”lost” competition may drastically decrease the final quality of the neural renderer. We thus experiment with taking the experiment where BN layers are disabled for zoomed images, and disable execution and optimization of the discriminator only for zoom scaled batches; or only for FB scale batches (as a wild guess). This also means removing the Adversarial and Feature-Matching losses from optimization. We also conduct another experiment with removing these losses for the whole training.

**Collecting BN statistics with high momentum, only on FB scale** (Fig. B.46)

Another source of possibly corrupted running average statistics of Batch Normalization lay ers may be in the rate of exponential decay with which they are collected (also referred to as momentum). By default, the BN layers have a decay rate 0.1, meaning that statistics of a sin gle batch may still be significantly present in the running average of up to 30 next batches (e.g. (1 *−* 0*.*1)30 *≈* 0*.*04 = 4% of the values still present after 30 batches). We experiment with a higher decay rate: 0.2 (significant for *≈* 15 batches), 0.3 (significant for *≈* 9 batches).

**Processing multiple batches before making an optimization step** (Fig. B.48)

A fairly small batch size (8) may also cause instability of statistical properties within a batch. We cannot try a higher batch size because of a memory limit on the hardware during training. However, we experiment with simply performing an optimization step every *k* batches instead of every single batch. Although it is not equivalent to the higher batch size, because the BN statistics are anyway averaged and added after processing every batch, the affine parameters are optimized using an average gradient, which may include information from a wide range of zoomed scales and body parts. The same applies to updating of all the learned parameters in the architecture, including the neural texture, convolutional layers of the neural renderer and discriminator. We try making the optimization step every 2 or 4 batches, as well as updating only discriminator every 2 batches (a technique proposed for improving GAN performance in [33]).

48

**Collecting BN statistics only on FB scale, fine-tuning only on FB scale** (Fig. B.49, B.50)

We combine the experiments on collection of statistics of BN layers only on full-body scaled images, and fine-tuning it after a certain epoch to only FB scale. Thus, during fine-tuning the statistics are collected from all the training data. We also try lower the learning rate of the neural texture, either decreased by 10 times, or by 1000 times. It has a motivation of preventing drastic updates of the neural texture, after is captured fine-grained details from the zoomed scales. The fine-tuning should restore the original FB quality, and possibly suppress the artifacts resulted from the zoomed training.

**Non-uniform distribution of augmentation zoom scales** (Fig. B.51)

Previously, we applied the camera-space augmentations by uniformly sampling a random zooming scale within a specified range. We later hypothesize, that it may be undesireable, as the FB frames will be less likely to be shown, e.g. for a range of scales between 0.7 and 3.5 (which is a moderately high zoom), the FB range is only within 0.7-1.1 scales, which is only 15% of the whole range. Hence, it is pointless to expect high quality of FB images. Moreover, we had a quick debugging experiment on learning with ever so rarely shown zoomed frames, and it hinted that adequate zoomed images can be learned, even if they are shown rarely. In this group of experiments, instead of a single range of scales with a uniform sampling, we use multiple bands of ranges, each having a weight. By changing the weights and ranges size, we can make some range of zooming scales more probable than the others. We use it to train on mostly FB images, with a gradually decreasing probability of higher zoom scales, up to *×*12 scale.

**Using the same scale of zoom augmentations within a batch** (Fig. B.52)

Following the hypothesis of incompatible statistics of zoomed frames, which may cause in stability of training with BN layers, we try to restrict every training batch to have the same scale of zooming, the same affine rotation rotation and the same center of camera-space augmentations, thus making the zoomed frames statistics as much comparable as possible.

**Disabling Adam optimizer’s first momentum** (Fig. B.53)

After experimenting with collection of statistics for the BN layers from FB scaled images, we suspected that the Adam optimizer’s momentum may occasionally push the optimization towards a certain harmful direction for many batches, due a slow decay of the running exponential averaging (0.9 by default). We experiment to check this suspicion, by setting the first momentum to 0.0, for the neural texture, or for the neural texture plus the discriminator.

49

**Strong affine augmentations** (Fig. B.55)

As a regularization technique, we experiment with increasing the variance of affine augmen tations. We apply a very strong translation (so that sometimes the avatar body parts may be just barely in the input frame), and very strong rotation (used to be *±*45*◦*, and we try *±*90*◦*). This should reduce both overfitting and discrepancy when rendering body parts at different screen locations, e.g. the image’s middle, borders, corners. The rotation additionally can improve inference quality on mobile, as in AR it’s possible to rotate the mobile device and thus observe the avatar horizontally or even upside down, for which the neural renderer is unlikely to be trained.

**Gradient clipping** (Fig. B.56, B.57, B.58)

The next regularization approach we use is gradient clipping, that should reduce optimization steps from gradient tensors that have too high norm values, and if such steps were made, it could lead us to an unpredictable location on the optimization landscape. Thus the gradient tensors with an exceeding norm value are normalized to a certain limit. It may also help to prevent overfitting towards training samples which are outlying or invalid, since they are a minority among the training samples, the clipped optimization steps are unlikely to hurt the training process.

We try to clip gradients of both the neural renderer and discriminator to norm 5.0, or to 2.0. Alternatively, we find a mean norm independently for the neural renderer and discriminator, and use it multiplied by either 1.0, 0.5 or 0.2 as a gradient clipping limit. It should effectively act as a lower learning rate, since the majority of gradients will be clipped. However, we also expect appearing of much fewer overfitting artifacts. We also experiment with adding a decaying learning rate scheduler, with a warmup period that sets up to 10*−*2learning rate to the neural renderer, and gradually decays to learning rate 10*−*6. We add the gradient clipping with an intuition, that it will help to preserve stability of optimization during the warmup with too high learning rate.

**Weight decay** (Fig. B.59, B.60, B.61, B.62, B.63, B.64)

Another regularization approach that we explore is weight decay (also referred to as L1 regu larization) on the learned parameters across the whole architecture. It should prevent overfitting by slowly pushing the weights tensors towards zero L1 norm. Thus, it decreases a chance of learning very high-magnitude weights, which is a common symptom of overfitting and instability of out puts. There is a known issue with Adam optimizer that it does not implement the weight decay in a mathematically correct way, thus we use AdamW [64] modification that solves it. We experiment with weight decay of neural renderer’s parameters to norm values *{*10*−i}*6*i*=0. Together with the found promising weight decay value of 10*−*2for the neural renderer, we experiment with weight decay of the neural texture and discriminator with the following values: (10*−*1*,* 10*−*1), (10*−*1*,* 10*−*2),

50

(10*−*2*,* 10*−*1), (10*−*2*,* 10*−*2). We also try weight decay 10*−*3 value for neural renderer alone; for neu ral renderer plus discriminator; and for neural renderer, discriminator and neural texture together.

**Dropout for channels of activation tensors** (Fig. B.66, B.67, B.68)

We try to add regularization to the neural renderer via 2D dropout layers [94] to randomly zero-out (completely) a few random channels of activation tensors. We experiment with adding dropout with probability *p ∈ {*0*.*1*,* 0*.*05*}* after all levels of either encoder or decoder or them both. We also try to apply it after every single convolution in the neural renderer, with *p ∈ {*0*.*1*,* 0*.*15*,* 0*.*2*}*. In theory this should force learning of multiple slightly similar convolutional fil ters that would act as backup to fit the training images in a presence of dropout, and then to provide stability of outputs during inference.

**Additive noise to neural texture’s initialization** (Fig. B.69)

The neural texture’s initialization with spectral mesh coordinates (see Figure B.1) may con tain values with a large magnitude. We hypothesize, that it may be an issue for body parts may be never shown on the ground truth images (bottom of shoes, armpits, top of the head). The texture pixels that correspond to those parts will never be changed away from the default initialization. Thus, after optimization, the contained values may be well beyond a typical range of values, caus ing the neural renderer to produce unstable results. We experiment with adding a random Gaussian noise (with *σ ∈ {*0*.*1*,* 0*.*025*}*) to the texture intialization, with an idea to partially randomize the unseen body parts on the texture, and reduce the appearing artifacts (see also Figure B.9). We also try a random neural texture intialization, to understand how convergence of the neural renderer depends on the baseline initiliazation, and whether the similar artifacts emerge.

**Additive noise augmentation for input and ground truth data** (Fig. B.70, B.71, B.72)

We try to regularize the neural renderer by applying a random additive Gaussian noise to the input frames (with *σ ∈ {*0*.*01*,* 0*.*02*,* 0*.*1*}*), i.e. affecting appearance of both the background and the body mesh on a frame. Alternatively, we add it to the neural texture before input frame rasterization (with *σ ∈ {*0*.*01*,* 0*.*02*,* 0*.*1*}*), i.e. affecting only the body mesh. Lastly, we try adding it to the ground truth images (with *σ ∈ {*0*.*05*,* 0*.*1*}*). This might make the network overfit less to subtle misalignment of body mesh fits, and instead encourages to save similar information in multiple parameters of the neural texture or the neural renderer.

**Reducing the number of parameters in the neural renderer** (Fig. B.65)

It is a common practice in the encoder-decoder based DNN models to have a much bigger number of learned parameters in the encoder rather than in the decoder [23, 83, 13]. This is because the encoder’s task of data compression into a smaller decodable size is harder than decompress

51

ing it. Still, we experiment with reducing the number of learned parameters in the encoder, by discarding the last convolution of the last level of ResNet18, effectively reducing the number of parameters in the neural renderer by 20%. We also try to reduce the number of output channels for all decoder levels by half, reducing the number of neural renderer parameters by 25%. This may act as regularization, that prevents overfitting.

**Reducing the number of parameters in the neural texture** (Fig. B.73, B.52)

The default neural texture has multiple downscales learned jointly with the neural texture in the original resolution. We referred to it as a multi-scale neural texture. We experiment on collect ing statistics of BN layers only from FB scaled images, but also training without those downscales of the neural texture (i.e. only 512 *×* 512 pixels resolution). Perhaps, the lowest resolution down scales of the texture may learn values, that when upsampled and added to the original resolution may change the values in the pixels, that correspond to the body parts that may not be seen on the ground truth images. This can cause values on the rasterized input frames that are abnormal for the trained neural renderer, and thus the visual artifacts with failed segmentation that we encountered.

We also try to take a standard experiment with zooming to joints, and then reducing the number of neural texture channels from 16 to 8. With that we try to understand importance of pixel-wise depth of the encoding for different body parts, and how it affects the visual quality.

**Pass downsampled input to levels of the neural renderer’s encoder** (Fig. B.74)

There are a few successful attempts in the literature [83, 13] for the generative models with an encoder-decoder architecture, to pass downsampled input frame to the levels of the encoder. This may reduce an effect of data propagation through the model with irreversible accumulation of errors. By reminding the encoder levels how the input data looks like, it may improve compression of the input into the values of bottleneck activations. Thus the learning process may be more stable. We experiment with replacing a few channels of intermediate activations with the downsampled input, which preserves the performance of inference (for possible execution on mobile devices). Alternatively, we try concatenation with intermediate activations instead, and thus having more learnable parameters.

**Adjust weights of training losses** (Fig. B.75, B.52)

The baseline architecture uses specific weights for training losses. Namely, Adversarial and Feature-Matching losses contribute with about 25 times lower weight. We try to set all the weights to 1, with motivation that the architecture will start satisfying all the losses, without the need for hyperparameter tuning to different training sequences. Although the magnitude of different losses may be incomparable, we believe, that ever since the Batch Normalization is used in most of DNN architectures nowadays, the specifically chosen loss weights matter less and less.

52

We also try to improve the segmentation artifact appearing in the experiments where BN statistics are collected from FB scaled frames. We try to double the segmentation (Dice) loss and compare the effect.

**Use fewer training samples** (Fig. B.76, B.50)

In the previous experiments we frequently encounter overfitting due to training on zoomed scaled images. We experiment, how the number of training samples affects the training. With too many samples there is a chance that the architecture will generalize well to all of them. On the other hand it may fit well all of them except for a few, that are too complex or invalid (having bad segmentation or SMPL-X fit in the dataset). Thus, these frames will be dominant in the magnitude of the losses, forcing the architecture to overfit to them no matter what, thus the artifacts (see Figure B.13b). We try to reduce the number of training samples from 397 to 27, that were manually selected to show the person from all the sides, and most of the adequately fit poses. We also try use the original training sequence, but manually removing a handful of frames that are fit badly by the neural renderer in most of our experiments (e.g. the ones containing change of lighting such as on Figure B.14 to the left).

**Learned MIP-maps of the neural texture** (Fig. B.77)

In the baseline architecture the library Nvdiffrast [57] for differentiable rasterization allows applying MIP-maps (see Figure B.6a) to the neural texture. However, the algorithm they use to generate the downscaled levels of the texture does not include any filtering, instead it only averages 4 neighboring pixels. The library allows to pass custom tensors as the MIP-maps, thus we try to make these tensors as learned parameters. Our motivation is that in those MIP-maps may be learned information for adjusting the rendering with respect to scale of zoom, since the far-away and full body scales will be rendered with low-resolution MIP-maps, while the zoomed frames with high resolution MIP-maps plus the main texture. We initialize the MIP-map tensors with a downscaled version of neural texture’s initialization. We do not synchronize the values learned in the neural texture and its MIP maps anyhow. To check the effect of the default MIP-mapping algorithm, we try disabling usage of any MIP-maps whatsoever.

53

**Chapter 4**

**Results and Discussion**

In this chapter we discuss quantitative and qualitative results from the mobile application and experiments with the baseline DNN; and we make conclusions about the researched approaches. Section 4.1 covers performance and visual quality of neural rendering in the mobile application. Section 4.2 compares DNN experiments and the impact of the attempted adjustments. Section 4.3 outlooks future research directions to improve the project.

**4.1 Mobile application results**

We compare performance of inferring the baseline DNN at different stages of progress dur ing completion of this thesis project. In the next paragraphs we will analyze Table 4.1. As was mentioned in Section 3.1, we can measure frame time of the AR application, which includes time of SMPL-X inference, input rasterization, passing it to the DNN, inferring the DNN and rendering the AR content. Alternatively, in our ”benchmarking” application mode, we infer the DNN for 5 minutes on the same input data to gather fair statistics on the inference time and maximum increase of the device temperature. However we noticed, that since we initialize SNPE with ”Sustained High Performance” profile, the maximum temperature is controlled by the device and it never ex ceeds 55*◦*C, thus we do not report it. As per the project task, we consider performance as real-time, if application frame time is below 33 ms, which corresponds to above 30 FPS.

In our initial implementation, when we encountered the limitation of OpenGL for rasterizing images of at most 4 channels, we generated the DNN’s input tensors using multiple OpenGL ras terizations of size *H × W ×* 4. Then the DNN was extended from the PyTorch side to accept these multiple rasterizations, and permute the data in memory to get the necessary *H × W × C* before starting the inference. Although the permutation would take place on the GPU or DSP, the perfor mance was fairly low even for resolution 256 *×* 256 pixels with 8 neural texture channels (row 1 in Table 4.1). Trying to run this layout for the desired resolution 512 *×* 512 and 16 channels, the performance would drop below 25 FPS (row 2). However, by implementing the quantization and packing of all neural channels with a single OpenGL rasterization and getting the input tensor in the

54

exact memory layout as required, the performance immediately improves for all such resolutions (rows 3,4,5) and becomes real-time on DSP. We could further increase the resolution of rendering, still having real-time performance with 640*×*640 pixels (row 6). To test the limits of the hardware, we try 1024*×*1024 resolution (row 7), which is obviously beyond real-time, as computations scale quadratically with resolution. On the bright side, the architecture was successfully converted for execution on DSP. In our experience, some neural layers may not work on DSP for high resolutions due to a hardware limit on tensor spatial dimensions or number of channels. Thus practically, such high resolutions are possible on mobile with the architecture we use, yet the performance is the bottleneck.

Next, following the experiments on the DNN’s training procedure, we try to run a modified architecture, with all Batch Normalization (BN) layers replaced with either Instance Normalization or Group Normalization layers. BN layers typically store running exponential average of batch statistics from the training data, thus allowing to use the pre-computed mean and standard deviation for each tensor, and normalize tensors with little to no overhead during inference. Instance and Group Normalization layers, however, by default require computing the statistics on each inference. Thus it is expected of them to have lower performance. As we measure, compared to BN layers, an architecture with Instance Normalization layers is about 30-50% slower (compare rows 5 and 6; rows 9 and 10), but still borderline real-time. Group Normalization layers are computed very inefficiently (row 13). In fact, documentation of SNPE does not mention support of it, thus it may be computed with hardly any hardware acceleration.

Finally, we examine how different encoder types of the neural renderer affect the perfor mance. While the baseline DNN uses ResNet18 [37], it might be expected for light-weight archi tectures like MobileNet [82, 43] or EfficientNet [91] to drastically outperform. However, from out attempts MobileNetV2 is not inferred faster for the same resolutions (rows 9,10,11), even though it has twice as fewer parameters. We speculate, that the depth-wise convolutional layers used in MobileNetV2 have no efficient implementation on the hardware. After replacing BN layers with Instance normalization layers, we observe the similar percent of performance degradation (row 10). MobileNetV3 [43] appears to be a bit faster for resolution 256 *×* 256 (row 14), however we could not convert models with any higher resolution into the SNPE format. We tested out EfficientNet Lite0 model, and it performs the best for resolution 512 *×* 512, while having a comparable number of parameters to MobileNetV2. However, inference on DSP results in a failed segmentation pre diction (see Figure B.16f). This artifact does not appear with GPU execution, thus it should be from high value range of tensors, which leads to strong inaccuracies due to quantized computations on DSP. Even so, the visual quality of the avatars with this encoder is below acceptable, thus we did not experiment with it any further.

55

| **Neural Renderer configuration** | | | **Input Layout** | **Application avg. frame time (ms)** | |  |
| --- | --- | --- | --- | --- | --- | --- |
| Encoder  type | Normali  -zation | Parameters number | on GPU | on DSP | on GPU |
| ResNet18 Batch 14.3M | | | 2-256-256-4 | 55–57 19–23 | | 49.6 |
| ResNet18 Batch 14.5M | | | 4-512-512-4 | 273–281 41–48 | | 270.5 |
| ResNet18 Batch 14.3M | | | 1-256-256-8 | 41–43 16.6 cap | | 39.1 |
| ResNet18 Batch 14.5M | | | 1-512-512-8 | 112–136 21–22 | | 115.7 |
| ResNet18 Batch 14.5M | | | 1-512-512-16 | 175–184 24–25 | | 173.7 |
| ResNet18 Instance 14.5M | | | 1-512-512-16 | 263–275 30–32 | | 248.3 |
| ResNet18 Batch 14.6M | | | 1-640-640-16 | 374–389 30–32 | | 364.5 |
| ResNet18 Batch 14.8M | | | 1-1024-1024-16 875–920 | 71–73 | | 863.9 |
| MobileNetV2 Batch 6.6M | | | 1-512-512-16 | 201–248 27–35 | | 206.5 |
| MobileNetV2 Instance 6.6M | | | 1-512-512-16 | 220–244 44–47 | | 221.4 |
| MobileNetV2 Batch 6.5M | | | 1-512-512-8 | 176–236 25–29 | | 188.2 |
| 12 EfficientNet-Lite0 Batch 6.1M | | | 1-512-512-16 | 134–142 16.6 cap 133.8 | |  |
| ResNet18 Group 14.5M | | | 1-512-512-16 | – 1228 | | – |
| MobileNetV3 Batch 6.7M | | | 1-256-256-8 | 37–38 16.6 cap | | 35.8 |
| MobileNetV3 any – | | | any higher |  | unsupported on hardware |  |

**Benchmarked avg.**

**#**

**inference time (ms)**

on DSP

1 19.3 2 37.9 3 3.2 4 14.1 5 16.4 6 23.2 7 21.4 8 57.6 9 15.5 10 37.1 11 13.6 7.1

13 1198.3 14 3.7 15

Table 4.1: Neural Renderer’s inference performance, reported for different encoder types, memory layout of input data, normalization layers. The timings are profiled on either mobile GPU or DSP computing units. The input layout is specified by 4 numbers from left to right: number of OpenGL textures where input frame is rasterized, frame height, frame width, effective number of neural channels stored in a single texture. Since OpenGL textures always have 4 channels, the last num ber bigger than 4 indicates that quantized 8-bit values are packed into a single texture channel to achieve a contiguous memory layout. The performance of ”Benchmarked inference time” columns is measured by continuously inferring the DNN with constant random input data, thus excluding: mesh inference, rasterization, output rendering in AR, overall mobile interaction. The application performance is measured at a single moments as an exponential average of last 20 frames. The minimum-maximum range of it is reported. The application performance has a software cap of 60 FPS (16.6 ms). Instance normalizations may be up to twice as slower than Batch normalizations, since every time statistics of each tensor have to be computed. Other normalization types lack support on the hardware, as well as architectures, such as MobileNetV3 [43]. The performance was measured on a smartphone Samsung S21+ with Qualcomm Snapdragon 888 SoC. Frames-per second value can be computed as FPS =1000ms

frame time ms.

56

Additionally to measuring the frame-time of different inference configurations, we also look at output of Qualcomm Snapdragon Profiler software, which can show a graph of hardware metrics, such as utilization of computing devices, memory caches and threads, as well as a number of cache misses and stalls by the threads of execution. On Figure B.19 we can see execution of a model with inefficient layout of inputs for the neural rendering DNN, with a necessary permutation of the data before inference (the configuration from row 4 of Table 4.1). On the figure, the plot titled ”CDSP % Utilization” (the first orange from the top) shows moments of intensive DSP computations, cor responding to the DNN’s inference. There are gaps with 0% utilization that indicate latency of preparing the next input frame. Notably, during the gaps we do not observe the intensive compu tations on GPU (red, yellow, green and purple plots), instead they operate in parallel, i.e. GPU and DSP do not need to synchronize between each other. However, due to the inefficient layout, there is a high rate of data cache misses in the DSP (the third plot from the bottom titled ”L2 DU Read miss”), related to necessity to permute segments of the data that are far apart in the memory. Thus, the cache lines that could be used in further computations may instead be overwritten by some of the permuted data, requiring to load the same memory again. From the timings we discussed, such inefficiency may lead to twice as increased inference time.

On Figure B.20, the layout of DNN’s inputs is optimal and does not require additional permu tations. As we can see, the rate of data cache misses (plot ”L2 DU Read miss”) is also considerably lower and about 0 most of the times. This means, that when a block of data enters the cache, nothing overwrites it. This by far reduces the latency of memory access, as repeated access to the data can be done directly from the cache (a cache ”hit”). We hypothesize, that this improvement on the rate of cache misses made the DNN inference up to twice as faster in the mobile applica tion. On Figure B.21, we use the same efficient input generation, but the neural rendering DNN has a MobileNetV2 encoder. Comparing the profiler output for it and the ResNet18 based models, we can notice a much smaller rate of cache misses when writing the data in memory (the second from the top orange plot ”DCache Store miss”). This indicates, that MobileNetV2 disentangles and parallelizes computations in such a way, that a single thread of execution exclusively writes to a single area of memory. Thus, almost no inter-cache synchronization has to occur when mul tiple threads dump data to the main memory. Even with this difference, we do not observe much of performance improvement from the MobileNetV2 encoder. This again hints on possibly inef ficient implementation of depth-wise convolution layers by the target DSP hardware. Lastly, we compare a ResNet18 based neural renderer with Instance Normalization layers used instead of BN layers (see Figure B.22). Again, these layers exhibit an excessive memory access in order to com pute statistics of tensors. Thus, the increased rate of cache misses can be observed on the plot ”L2 DU Read miss”. Furthermore, notice a much bigger intensity of DSP processing reflected on the

57

”Packets executed” plot, meaning that the sheer number of additional computations may cause the 30-50% slower performance of Instance Normalization layers.

Summarizing the performance analysis of different configurations, the speed of inference on DSP surpasses GPU by far. In fact, for any resolution, the application frame time with GPU in ference is not real-time, while DSP is real-time up to resolution 640 *×* 640, which is higher than we originally aimed for. Using different encoders or normalization layers for the neural renderer proven to generally hurt the performance on mobile. It is thus advisable to stick to resolution 512 *×* 512 pixels with a ResNet18 based encoder and Batch Normalization layers, as it provides comfortable and smooth AR experience. However in the future, if MobileNet would receive more of hardware support (perhaps an application-specific integrated circuit for depth-wise convolu tions), then it would probably outperform ResNet18 a lot.

Examples of the mobile application in use can be found on Figure B.16. In the application, a user can choose a specific DNN, animation and settings of inference. We also compare how different projection modes affect the AR experience: with the default screen-wide projection, dy namically cropped projection around joints and the dynamically cropped projection with clipping to screen boundaries (as on Figure B.10 in the respective order). When observing the avatar from afar, the default projection results in inferring the neural renderer with a very small rasterization of the body mesh and DNN might not generalize well for such small scales (Figure B.17a). By cropping around joints (clipping yields the same projection), we can generate input images with a full-body scaled avatar always, thus DNN produces meaningful and stable images regardless of the distance. On Figure B.17b we can see the generated image, and Figure on B.17c the extents of the crop. In the scenario of zooming in, the screen projection results in an image that is rendered partially, thus wasting some spent computations (Figure B.18a). The dynamic cropping around joints surrounds the avatar a bit tighter, thus appearing sharper (Figure B.18b). By clipping this projection to screen boundaries, we get the absolute minimal projection, which also allows to feed the DNN with a zoomed image, and given that the network is trained to produce such images, the quality improves a bit more (Figure B.18c).

We also compare images, that were generated from the same input on either mobile GPU, mobile DSP, or desktop GPU computing devices (in the respective order on Figure B.15). You can notice, that the mobile GPU and desktop GPU quality doesn’t differ much, even though the former was inferred from an input rasterization with quantized INT8 values, and the latter with FP32. On the other hand, when inferring on DSP with all intermediate computations quantized, the output image has discrete transition of colors (especially seen on the red area of the T-shirt), and overall the image appears noisy. On the other hand, no major artifacts appear that would dramatically hurt

58

the visual quality. Given that DSP allows real-time inference, the achieved quality-performance tradeoff seems advantageous.

Lastly, we compare rasterization of the input tensors on mobile, with the plain full-resolution neural texture (Figure B.8a), or when MIP-maps and anisotropic filtering is applied (Figure B.8b). Both avatar images are inferred on mobile GPU. The difference should only be visible for triangles at steep angles to the camera, such as on the body contour, where triangles are more and more perpendicular to the camera view. However, we observe virtually no difference. We speculate, that the improvement is much less pronounced than on the example Figure B.7, because of a relatively small number of steep-angled triangles. Additionally, the input frames are processed with the neural rendering DNN, thus it is possible that any difference is further hidden behind the noise of internal computations inside the DNN.

**4.2 Training experiments results**

The main target of our experiments on training of the baseline DNN architecture for syn thesizing images of different scales (from far-away, to close-up), was to obtain admissible quality, that would not hurt the AR experience in the mobile application with sudden artifacts, as shown on Figures B.17a (far-away), B.23 (close-up). Secondarily, since generalization to multiple scales may act as a regularization, the issues related to either overfitting, or learning from not-perfectly aligned SMPL-X body model, or lacking fine-grained details in the full-body scale images could be partially solved. Using camera-space augmentations, and using the same baseline architecture to train only for FB scale, we already achieved improvement (Fig. B.23) in terms of validation metrics, and visually the images appear a sharper. This seems very logical, after observing the quality loss of the old approach of image-augmentations (Fig. 3.3).

B.2 B.3 B.4 B.5

B.9

B.13, B.14

**4.3 Future work**

The baseline model [36] is still state-of-the-art in its particular task – rendering of a full body avatar using information from a few monocular uncalibrated images, or a video sequence. However, as the original authors point out, it is an optimization-based approach that takes about half a day to converge, and may be impractical in a scenario where users can scan themselves and

59

after a short time be able to use their avatars. The training time can be improved by doing further research on meta-learning, where both architecture and its initialization are sought, so that fine tuning of it would converge to acceptable results after a very short optimization. On the other hand, a completely new algorithmic feed-forward solution could be proposed, so that a single inference would yield all the necessary data to generate realistic avatars.

Regarding this thesis project, the neural renderer’s inference performance could be further improved. For example, we could employ already described approaches of knowledge distillation or neural architecture search, to find an architecture with fewer parameters, but similar visual qual ity. However, such research efforts are known to be notoriously big, in terms of experimentation and amount of computations. On the other hand, purely algorithmic tricks could be used. The approach of pipelining [72] could be used, to split the network into independent stages, with the idea that as soon as a stage finishes computation on the given data, it could pass the data to the next stage, and at the same time accept new input for processing. In the context of the mobile execution, some parts can be executed on different computing units (GPU, DSP) and exchange the data using the shared memory. As was shown in the Section 4.1, in the current implementation of the mobile inference there is some idleness of DSP between frames, caused by latency of rasterizing each new frame and sending it to the DNN. More efficient organization of calculations could be researched to minimize this idleness, potentially gaining about 10-15% of performance. For example, CPU multithreading could be better utilized in the future, or batch inference of the DNN on mobile to improve throughput of data through the computing units.

Improving the quality of synthesized images is also a big field for research. Making small training procedure adjustments as was described in this thesis proved to be ineffective to solve all the current issues with overfitting and randomness. Thus, research on other architectures is yet again more preferable. This also requires to decide on generalization ability of the rendering architecture. On one hand, sticking to one-DNN-one-avatar approach, allows to obtain the full realism for the particular person. A lot of attention is brought to NeRF [66, 14] based methods, from which very realistic images can be sampled from many novel views. On the other hand, it would be beneficial if a single DNN could be used to render many avatars, distinguishing them by neural textures. Thus we could place multiple different avatars in a single frame of AR, and generate images of all of them at once.

Such scenario anticipates the dawn of true telepresence communication, given that mobile devices are already capable of doing real-time DNN inference. However, this would also require to overlook the cropping approach that was used in this thesis to get higher and stable visual quality. Imagine if we were to render two avatars separated by a considerable distance. By using a bounding box that includes both of them to do frustum cropping, both avatars will be quite small in the frame,

60

once again bringing up the necessity for the DNN to be stable on far-views (as in Figure B.17a) One way to extend cropping to a multi-avatar scenario, is to use the fact that in OpenGL we can rasterize to a part of an image. Thus, we could compute bounding boxes for each avatar separately, then to distribute the input frame between avatars rasterizations, proportional to used image area in AR (See Figure B.11). An additional advantage is that we do not need to worry about avatars overlapping, because they will be rasterized one by one, without affecting each other. The time of rasterizing *N* avatars is still extremely smaller than a DNN inference time. The proportion of areas on the input frame can also be dynamically adjusted to yield better images for nearby avatars, and decreasing quality of far-away ones.

Lastly, the issue of stability in rendering unseen body parts has to be tackled. In this thesis we used a handcrafted solution to replace the corresponding texture parts with content that was seen by the neural renderer. However, we believe, that a more general algorithmic approach could be used, e.g. by inpainting the neural texture with logical content, or by feeding the renderer during training with images that show the missed body parts, even though the training sequence does not contain them. These could be images of other people, augmented to match the hair or clothes colors. Or these could be synthesized from some other model that saw these body parts.

61

**Chapter 5**

**Conclusions**

In this Master thesis, we firstly implemented an Augmented Reality application for Android mobile devices. It allows to detect surfaces in the camera view and place full-body human avatars, as if they stand on a surface for real. The correct images of the avatars are synthesized by a DNN, which was specifically integrated to work natively on the mobile device hardware, in real-time. The integration includes inference of a human body 3D model for a certain body pose, and pro jection of it as an image of the neural network’s inputs. By means of quantization, data layout and parallel GPU processing, we arranged an efficient way of generating the input data. As a re sult, the input data can be directly forwarded to the DNN processing, without additional memory reordering required. Using Qualcomm Snapdragon hardware capabilities with Digital Signal Pro cessor computing unit, we achieved real-time DNN inference with latency well below the minimum requirement and sufficient resolution of the synthesized images. Even bigger resolutions are avail able, still within real-time performance. We propose to use an algorithmic approach of dynamic cropping of the physical camera’s view frustum, to always render input images with avatar being at the center and with only the body parts that will be visible in the current view of AR. This led to automatically better image quality when looking at the avatar from far away.

Secondly, we carried out research on improving the training procedure of the baseline DNN, to achieve higher quality of images that can be encountered in the AR scenario. Using the nature of the DNN’s input data, we propose to use camera-space augmentations to train the neural network on zoomed images. We studied how such training can improve stability of synthesized images to minor shifts of body pose or viewpoint. On the other hand, it may also reduce original visual quality of full-body images generated by the baseline, or amplify visual artifacts related to overfitting, discrepancy of statistical properties on different scales of zooms, and out-of-distribution rendering of body parts that were never seen during training. We tried to overcome those issues using well known regularization techniques, as well as our own insights. In attempt to preserve the original DNN architecture, the effect of the researched adjustments proved to be ever so slight. Many of them improve convergence speed, yet speeding up emergence of the overfitting artifacts. Although we had a few successful experiments that partly solve the issues, they come at a cost of being

62

less efficiently implemented on the mobile hardware, leading to drastic decrease in the real-time performance.

We conclude this work, declaring that all the project tasks were completed and the goal of the Master thesis was achieved. In the future, we strive to solve the remaining visual artifacts by over looking and significantly deviating from the baseline architecture, using the knowledge obtained during this work. The mobile AR experience is also subject to expansion towards rendering of multiple avatars by a single neural network. Also, since the Qualcomm Snapdragon hardware plat form we used can also be found in other mobile devices, such as AR glasses, we expect to integrate the full-body avatars here as well. All the efforts that we made prove that the embedded real-time artificial intelligence is achievable for the state-of-the-art hardware. However, it requires careful and efficient integration into the products. The approaches used in this work may be relevant to projects on AI-based immersive telecommunication, Augmented/Virtual Reality, user-device in teraction, real-time processing of audio/video, etc.

We want to once again emphasize the negative impact of inefficiencies during storing or reading of data in computer memory. If the data layout is too sparse or accessed in non-sequential patterns, it may easily reduce the performance by half, as we have shown in this work. It can bewil der software developers and wrongfully convince that a task is unfeasible for real-time execution. Regarding the quality of DNN results, sometimes it’s more rewarding to think of an algorithmic solution to the problems, rather than trying to make the DNN generalize all the corner cases. For example, we applied that to improve far-away image quality in AR, by simply never passing the far-away inputs to the DNN to begin with.

63

**Acknowledgments**

This thesis project was completed on premises of Samsung AI Center at Moscow, in direct collaboration with its lab of Visual Understanding. I’d like to express my gratitude to all its mem bers, and personally Renat Bashirov, Ilya Zakharkin, Evgenia Ustinova, Artur Grigoriev, David Svitov, Aleksei Ivakhnenko for sharing their knowledge and supporting the research.

I thank my research advisor Victor Lempitsky for directing the project and giving such in valuable networking and research opportunities.

I thank Skolkovo Institute of Science and Technology, for being the best place for education and research in Russia. I’ve learned thrice as much as I expected here, got to know the brightest people of the country, and jump started my own career before even graduating.

**Author Contribution**

The developments presented in this thesis, are a part of the bigger project of Samsung AI Center at Moscow on realistic full-body human avatars. The presented advancements are based on the project’s state as of June 2021.

The described mobile application for Android OS has been developed from scratch solely by the author (using the aforementioned software and hardware products). This includes, but not limited to the following:

• auxiliary code for Android application interaction;

• passing camera frames through 3D tracking library;

• implementing the human body model [74] inference in Java;

• preparing the DNN models written in Python for execution as SNPE modules (code adapta tion, binaries conversion, DNN quantization);

• implementing low-level GPU programs for real-time generation of the DNNs’ input data; • implementing frustum cropping w.r.t. camera angle relative to the rendered avatar position.

The research on avatars quality in both far-out and close-up scales has been carried out in isolation from research directions of the other lab members. All the presented experiments with the neural network training were implemented by the author. Among other things this includes: implementation of the PyTorch module for camera-space affine augmentations, retrieval of demon strative images for the appendix. Many of the conclusions and further ideas were suggested by the thesis supervisors and the lab members.

64

**2D** Two Dimensional **3D** Three Dimensional **ADB** Android Debug Bridge **AI** Artificial Intelligence

**Abbreviations**

**LR** Learning rate

**MAC** Multiply-Accumulate operation **MAE** Mean Absolute Error

**ML** Machine Learning

**API** Application Programming Interface **AR** Augmented Reality

**BCHW** Batch-Channel-Height-Width **BHWC** Batch-Height-Width-Channel **BN** Batch normalization

**CNN** Convolutional Neural Network **CPU** Central Processing Unit

**DNN** Deep Neural Network

**DSP** Digital Signal Processor

**FB** Full-body

**FM** Feature matching (loss)

**FP16** 16 bits floating point number **FP32** 32 bits floating point number **FPS** Frames per second

**GAN** Generative Adversarial Network **GLSL** OpenGL Shading Language **GPU** Graphics Processing Unit

**IN** Instance normalization

**INT16** Integer number of 16 bits

**INT32** Integer number of 32 bits

**INT8** Integer number of 8 bits

**KIMGS** Kilo-images, 1000 images **L1** Synonym of Mean Absolute Error **L2** Synonym of Mean Squared Error

**MLP** Multi-layer perceptron

**MS (ms)** Milliseconds

**MS-SSIM** Multi-Scale Structural Similarity **MSE** Mean Squared Error

**NPU** Neural Processing Unit

**ONNX** Open Neural Network Exchange **OpenCL** Open Computing Language

**OpenGL** Open Graphics Library

**PQT** Post training quantization

**PSNR** Peak Signal-to-Noise ratio

**QAT** Quantization-aware training

**RGB** Red-Green-Blue (pixel channels) **RGBA** Red-Green-Blue-Alpha

**RGBS** Red-Green-Blue-Segmentation

**SDK** Software Development Kit

**SMPL** Skinned Multi-Person Linear model [62] **SMPL-X** SMPL - Expressive [74]

**SNPE** Snapdragon Neural Processing Engine **SSIM** Structural Similarity

**SoC** System-on-a-chip

**TPU** Tensor Processing Unit

**UB** Upper-body

**VGG** Neural architecture proposed in [85] **VR** Virtual Reality

65

**Appendix A**

**Relevant mobile application code**

In this appendix we present listings of code that are the most relevant to realtime input gener ation of neural rendering DNN on a mobile device. We omit about 85% of the code related to setup of the Android user interface, main loop, logging, loading of assets and configuration, interaction with the user, initialization of OpenGL components, selection of features, animations, rendering of the camera frame, tracked planes, etc. We also do not find it necessary to publish the full Python code for DNN training, since the adjustments presented in the Section 3.3 are reproducible from the description.

Listing A.1: Generator.java

Comment: The Generator class uses SNPE neural network API to map certain pre-allocated data buffers as locations of DNN input and output. Thus, on each inference no additional memory allocation is invoked. You can notice, that the network is created with multiple ”tries”, in order to try different combinations of security parameters for SNPE initialization. Without it, the network might fail to load required third party libraries.

**public class** Generator **implements** Closeable {

**private** SnpeNeuralNetwork mDNN;

**private static final** String INPUT\_NEURAL\_FRAME\_TENSOR\_NAME = "neural\_frame", *,→* INPUT\_UV\_FRAME\_TENSOR\_NAME = "uv\_frame";

**private static final** String OUTPUT\_TENSOR\_NAME = "rgbs";

5**private static** String mDlcFilename, mDlcCacheFilename;

**public** Generator(Application application, PipelineConfig config, **boolean** *,→* useRuntimeDsp) **throws** IOException {

mDlcFilename = *,→*

Paths.get(config.getGeneratorDlcAsset()).getFileName().toString();

10 String dlcFilenameNoExt = "";

**int** pos = mDlcFilename.lastIndexOf(".");

**if** (pos > 0) {

dlcFilenameNoExt = mDlcFilename.substring(0, pos); }

15 String dlcCacheFilenameNoExt = "CACHED\_" + dlcFilenameNoExt; mDlcCacheFilename = dlcCacheFilenameNoExt + ".dlc";

**final int** N\_TRIES = 8;

**for**(**int** tryOption = 0; tryOption < N\_TRIES; ++tryOption) {

20 **boolean** useDsp = useRuntimeDsp ? (tryOption < 4) : **false**; **boolean** useUnsignedPd = (tryOption % 2 == 0);

**boolean** checkUnsignedPd = (tryOption % 4 < 2);

AssetFileDescriptor asset = *,→*

application.getAssets().openFd(config.getGeneratorDlcAsset());

InputStream modelAsset = asset.createInputStream();

25 **int** modelAssetNBytes = (**int**)asset.getDeclaredLength(); **try** {mDNN = **new** SnpeNeuralNetwork(application, dlcCacheFilenameNoExt, *,→* modelAsset, modelAssetNBytes, useDsp, useUnsignedPd, *,→*

checkUnsignedPd);

**break**;

66

} **finally** {

30 modelAsset.close();

}**if** (tryOption == N\_TRIES - 1) {

**throw new** RuntimeException("Can't initialize SNPE with any *,→*

}

35 }

combinations of flags!");

}**public void** mapInputNeuralFrameBuffer(ByteBuffer nativeBuffer, *,→* PipelineConfig.QuantizationInfo quantization) {

mDNN.mapInputTensor(INPUT\_NEURAL\_FRAME\_TENSOR\_NAME, nativeBuffer, quantization.quantizedValueOfZero, quantization.quantizationDelta); 40 }**public void** mapInputUvFrameBuffer(ByteBuffer nativeBuffer, *,→* PipelineConfig.QuantizationInfo quantization) {

**if** (quantization == **null** || nativeBuffer == **null**) { **return**; }

mDNN.mapInputTensor(INPUT\_UV\_FRAME\_TENSOR\_NAME, nativeBuffer,

quantization.quantizedValueOfZero, quantization.quantizationDelta); 45 }**public void** mapOutputBuffer(ByteBuffer nativeBuffer) {

mDNN.mapOutputTensor(OUTPUT\_TENSOR\_NAME, nativeBuffer);

}**public** ByteBuffer getMappedNeuralFrameBuffer() {

50 **return** mDNN.getInputMappedBuffer(INPUT\_NEURAL\_FRAME\_TENSOR\_NAME); }**public** ByteBuffer getMappedUvFrameBuffer() {

**return** mDNN.getInputMappedBuffer(INPUT\_UV\_FRAME\_TENSOR\_NAME); }

55 **public** ByteBuffer getMappedRgbsFrameBuffer() {

**return** mDNN.getOutputMappedBuffer(OUTPUT\_TENSOR\_NAME); }**public** String getDlcFilename() {**return** mDlcFilename;}

**public** String getDlcCachedFilename() {**return** mDlcCacheFilename;} 60 **public void** forward() {mDNN.forward();}

**public** NeuralNetwork.Runtime getSnpeRuntime() {

**if** (mDNN != **null**) **return** mDNN.getRuntime();

65 }

**return null**;

@Override

**public void** close() **throws** IOException {mDNN.close();} }

Listing A.2: BenchmarkRunner.java

Comment: The BenchmarkRunner class shows how in-application benchmarking is implemented. We measure running average of frametime, as well temperature using a mobile sensor.

**public class** BenchmarkRunner **implements** Closeable {

**public static class** BatchStatistics {

**public long** MinMs = Long.MAX\_VALUE;

**public long** MaxMs = 0;

5 **public long** NFrames = 0;

}**public static class** BenchmarkStatistics {

**public long** StartupTimeMs;

**public long** BenchmarkTotalMs;

10 **public int** NFrames;

**public long**[] TimestampsHistoryMs;

**public long**[] NFramesHistory;

**public long** MinMs = Long.MAX\_VALUE, MaxMs = 0;

**public float** AvgMs = 0f;

15 **public float**[] AvgTimingHistoryMs;

**public long**[] MinTimingHistoryMs;

**public long**[] MaxTimingHistoryMs;

**public float** MaxTemperatureCelsius = Float.MIN\_VALUE;

**public float**[] TemperatureHistoryCelsius;

20 **public int** LastHistoryIndex = 0;

**public** BenchmarkStatistics(**int** capacity) {

TimestampsHistoryMs = **new long**[capacity];

AvgTimingHistoryMs = **new float**[capacity];

67

25 MinTimingHistoryMs = **new long**[capacity]; MaxTimingHistoryMs = **new long**[capacity];

TemperatureHistoryCelsius = **new float**[capacity];

}

30 }

NFramesHistory = **new long**[capacity];

**private** Generator mGenerator;

**private boolean** mIsBenchmarking = **false**;

**private** ByteBuffer mNeuralFrameBuffer;

35 **private** ByteBuffer mUvFrameBuffer;

**private** ByteBuffer mRgbsBuffer;

**private float** mCurrentTemperature;

**private long** mCurrentTimestampMs;

**private long** mBenchmarkBeginTimestampMs;

40 **private static final int** N\_BYTES\_PER\_CHANNEL = 1;

**private static final int** N\_RGBS\_CHANNELS = 4;

**public** BenchmarkStatistics benchmark(Activity activity, Consumer<Integer> *,→* batchCallback, PipelineConfig config, Generator generator, **int** *,→*

seconds, **int** measurementPeriodSeconds) **throws** IOException {

**long** startupTime = initializeGenerator(activity.getApplication(), config, *,→* generator);

45 BenchmarkStatistics stats = runAndBenchmark(seconds, *,→* measurementPeriodSeconds, batchCallback);

stats.StartupTimeMs = startupTime;

}

**return** stats;

50 **public boolean** isBenchmarking() {

**return** mIsBenchmarking; }

@Override

55 **public void** close() **throws** IOException {mGenerator.close();}

**private long** initializeGenerator(Application application, PipelineConfig config, *,→* Generator generator) **throws** IOException {

**long** beginTimestampMs = System.currentTimeMillis();

60 **if** (generator == **null**) {

mGenerator = **new** Generator(application, config, **false**);

} **else** {

mGenerator = generator; }

65 mNeuralFrameBuffer = ByteBuffer.allocateDirect(

getTensorSize(config.getSnpeNeuralFrameDims()) \* N\_BYTES\_PER\_CHANNEL)

.order(ByteOrder.nativeOrder());

mGenerator.mapInputNeuralFrameBuffer(mNeuralFrameBuffer, *,→*

config.getNeuralFrameQuantization());

70 mUvFrameBuffer = ByteBuffer.allocateDirect(

getTensorSize(config.getSnpeUvFrameDims()) \* N\_BYTES\_PER\_CHANNEL)

.order(ByteOrder.nativeOrder());

mGenerator.mapInputUvFrameBuffer(mUvFrameBuffer, config.getUvQuantization());

75 mRgbsBuffer = ByteBuffer.allocateDirect(

*,→*config.getSnpeNeuralFrameDims()[1]\*config.getSnpeNeuralFrameDims()[2] *,→* \* N\_RGBS\_CHANNELS \* N\_BYTES\_PER\_CHANNEL)

.order(ByteOrder.nativeOrder());

mGenerator.mapOutputBuffer(mRgbsBuffer);

80 **return** System.currentTimeMillis() - beginTimestampMs;

}

**private** BenchmarkStatistics runAndBenchmark(**int** benchmarkSeconds, **int** *,→* recordPeriod, Consumer<Integer> batchCallback) {

85 mIsBenchmarking = **true**;

BenchmarkStatistics stats = **new** BenchmarkStatistics(benchmarkSeconds / *,→* recordPeriod);

mBenchmarkBeginTimestampMs = System.currentTimeMillis();

68

90 **for**(**int** i = 0; i < benchmarkSeconds; i += recordPeriod) { batchCallback.accept(benchmarkSeconds - i);

**long** batchBeginTimestampMs = System.currentTimeMillis();

BatchStatistics batchStats = runFor(recordPeriod);

**float** batchTimingMs = (System.currentTimeMillis() - batchBeginTimestampMs); 95 **float** avgBatchTimingMs = batchTimingMs/(**float**)batchStats.NFrames; stats.NFramesHistory[stats.LastHistoryIndex] = batchStats.NFrames;

stats.NFrames += batchStats.NFrames;

stats.TimestampsHistoryMs[stats.LastHistoryIndex] = mCurrentTimestampMs; stats.AvgTimingHistoryMs[stats.LastHistoryIndex] = avgBatchTimingMs;

100 stats.MinTimingHistoryMs[stats.LastHistoryIndex] = batchStats.MinMs; stats.MaxTimingHistoryMs[stats.LastHistoryIndex] = batchStats.MaxMs;

stats.MinMs = Math.min(stats.MinMs, batchStats.MinMs);

stats.MaxMs = Math.max(stats.MaxMs, batchStats.MaxMs);

stats.AvgMs += batchTimingMs;

105 mCurrentTemperature = TemperatureMeter.getCpuTemperature(); stats.TemperatureHistoryCelsius[stats.LastHistoryIndex] = *,→*

mCurrentTemperature;

stats.MaxTemperatureCelsius = Math.max(stats.MaxTemperatureCelsius, *,→* mCurrentTemperature);

++stats.LastHistoryIndex; }

110 stats.AvgMs /= stats.NFrames;

stats.BenchmarkTotalMs = System.currentTimeMillis() - *,→* mBenchmarkBeginTimestampMs;

}

**return** stats;

115 **private** BatchStatistics runFor(**int** nSeconds) {

**long** batchBeginTimestampMs = System.currentTimeMillis();

BatchStatistics batchStats = **new** BatchStatistics();

**while** (**true**) {

**for** (**int** i = 0; i < 10; ++i) {

120 **long** beginTimestampMs = System.currentTimeMillis(); mGenerator.forward();

**long** elapsedMs = (System.currentTimeMillis() - beginTimestampMs);

batchStats.MinMs = Math.min(elapsedMs, batchStats.MinMs);

batchStats.MaxMs = Math.max(elapsedMs, batchStats.MaxMs);

125 ++batchStats.NFrames;

}mCurrentTimestampMs = System.currentTimeMillis();

**if** ((mCurrentTimestampMs - batchBeginTimestampMs) >= (nSeconds \* 1000L)) {

130 }

**break**;

}**return** batchStats; }

135 **private int** getTensorSize(**int**[] dims) {

}

}

**return** dims != **null** ? Arrays.stream(dims).reduce(1, (acc, val) -> acc\*val) : 0; Listing A.3: Frustum.java

Comment: The Frustum class illustrates how near/far/bottom/top/left/right parameters of a view frustum correspond to a projection 4 *×* 4 matrix.

**public class** Frustum {

**public float** left, right, bottom, top, near, far;

**public void** fillFromProjection(**float**[] projectionMatrix) {

5 near = projectionMatrix[14]/(projectionMatrix[10] - 1f); far = projectionMatrix[14]/(projectionMatrix[10] + 1f);

top = near/projectionMatrix[5]\*(projectionMatrix[9] + 1f);

bottom = near/projectionMatrix[5]\*(projectionMatrix[9] - 1f);

right = near/projectionMatrix[0]\*(projectionMatrix[8] + 1f);

10 left = near/projectionMatrix[0]\*(projectionMatrix[8] - 1f); }

**public void** toMatrix(**float**[] destination16) {

**float** inv\_rl = 1f/(right - left);

15 **float** inv\_tb = 1f/(top - bottom);

69

**float** inv\_fn = 1f/(far - near);

**float** doubleN = 2f \* near;

destination16[0] = doubleN \* inv\_rl;

destination16[8] = (right + left) \* inv\_rl; 20 destination16[5] = doubleN \* inv\_tb; destination16[9] = (top + bottom) \* inv\_tb; destination16[10] = -(far + near) \* inv\_fn; destination16[14] = -(far \* doubleN) \* inv\_fn;

25 } }

destination16[11] = -1f;

Listing A.4: ProjectionCropper.java

Comment: The ProjectionCropper class transforms a screen projection matrix into a crop, given crop center and height in space of camera coordinates.

| **package** com.saic.arcoreavatar.render.ml\_input\_output;  **import** android.opengl.Matrix;  **import** android.renderscript.Matrix4f;  **import** android.util.Log;  **public class** ProjectionCropper {  **private final** Frustum mOldFrustum = **new** Frustum();  **private final** Frustum mNewFrustum = **new** Frustum();  **private final** Frustum mNewFrustumRelativeNdc = **new** Frustum();  **private final static float** MARGINS\_SCALER = 1.0f;  **public** ProjectionCropper() {}  **public** Frustum getOldFrustum() { **return** mOldFrustum; }  **public** Frustum getNewFrustum() { **return** mNewFrustum; }  **public** Frustum getNewFrustumRelativeNdc() { **return** mNewFrustumRelativeNdc; } **public void** cropProjection(**float**[] projectionMatrix, **float**[] cropCenterInView, *,→* **float** cropHeightInView, **float** unstretchingScaler) {  mOldFrustum.fillFromProjection(projectionMatrix);  // Construct section of frustum  // 1. Shift near plane parallel to the location of avatar in view space // 2. Project shifted plane back onto original near plane  **float** projectOnNearPlane = -mOldFrustum.near/cropCenterInView[2];  **float** projectedCropX = cropCenterInView[0]\*projectOnNearPlane;  **float** projectedCropY = cropCenterInView[1]\*projectOnNearPlane;  mNewFrustum.near = mOldFrustum.near;  mNewFrustum.far = mOldFrustum.far;  **float** finalCropHeightInView = cropHeightInView\*unstretchingScaler;  **float** xMultiplier = MARGINS\_SCALER\*finalCropHeightInView\*projectOnNearPlane *,→* / mNewFrustum.near;  **float** yMultiplier = xMultiplier;  mNewFrustum.right = mOldFrustum.right \* xMultiplier + projectedCropX; mNewFrustum.left = mOldFrustum.left \* xMultiplier + projectedCropX; mNewFrustum.top = mOldFrustum.top \* yMultiplier - projectedCropY; mNewFrustum.bottom = mOldFrustum.bottom \* yMultiplier - projectedCropY;  // Compute relative parameters of new and old frustums  // (to construct mesh coordinates, on which infered image is rendered) **float** oldFrustumInvSpanX = 1f/(mOldFrustum.right - mOldFrustum.left); **float** oldFrustumInvSpanY = 1f/(mOldFrustum.top - mOldFrustum.bottom); **float** oldFrustumInvSpanZ = 1f/(mOldFrustum.far - mOldFrustum.near); mNewFrustumRelativeNdc.top = mNewFrustum.top \* oldFrustumInvSpanY; mNewFrustumRelativeNdc.bottom = mNewFrustum.bottom \* oldFrustumInvSpanY; mNewFrustumRelativeNdc.right = mNewFrustum.right \* oldFrustumInvSpanX; mNewFrustumRelativeNdc.left = mNewFrustum.left \* oldFrustumInvSpanX; Matrix.frustumM(projectionMatrix, 0, mNewFrustum.left, mNewFrustum.right, *,→* mNewFrustum.bottom, mNewFrustum.top, mNewFrustum.near, *,→*  mNewFrustum.far);  }  } |
| --- |

5

10

15

20

25

30

35

40

45

70

Listing A.5: NeuralProjectionCorrectionSelector.java

Comment: The NeuralProjectionCorrectionSelector class does projection matrix scaling, to make it render square images (as the neural network was trained), also it modifies the projection to vertically flip all images, to account to the difference of physical layout of pixels in OpenGL and SNPE.

**public class** NeuralProjectionCorrectionSelector {

**public** enum AdjustmentType {

UvMapToHeight,

UvMapToHeightCoord,

5 UvMapToZoomedWidth,

FrustumCrop,

FrustumCropZoom,

}**private static** AdjustmentType mCurrentAdjustment = AdjustmentType.FrustumCrop; 10 **private static float**[] CropCenterInView = **new float**[3];

**public static void** updateNeuralScreenCoords(PartialBackgroundRenderer *,→* partialBackgroundRenderer, **int** viewportWidthPx, **int** *,→*

viewportHeightPx, **int** virtualWidthPx, **int** virtualHeightPx, **float** *,→*

zoomFactor) {

**float** relativeExtentY = 1f;

**float** relativeExtentX = 1f;

15 **if** (mCurrentAdjustment == AdjustmentType.UvMapToHeight) { **float** magnification = viewportHeightPx / (**float**)virtualHeightPx;

relativeExtentX = magnification \* (**float**)virtualWidthPx/ viewportWidthPx; } **else if** (mCurrentAdjustment == AdjustmentType.UvMapToZoomedWidth) { **float** magnificationX = viewportWidthPx / (**float**)virtualWidthPx \* *,→*

zoomFactor;

20 relativeExtentX = magnificationX \* (**float**)virtualWidthPx / viewportWidthPx; relativeExtentY = magnificationX \* (**float**)virtualHeightPx / *,→*

viewportHeightPx;

}partialBackgroundRenderer.updateScreenCoords(relativeExtentX, relativeExtentY); }

25 **public static float** transformProjectionFitHeight(**float**[] projectionMatrix, *,→* Pair<Integer, Integer> screenResolution, Pair<Integer, Integer> *,→*

neuralFrameResolution) {

**if** (mCurrentAdjustment == AdjustmentType.UvMapToHeight) {

**float** toUniformScaleX = screenResolution.first / *,→*

(**float**)neuralFrameResolution.first;

**float** toUniformScaleY = screenResolution.second / *,→*

(**float**)neuralFrameResolution.second;

**float** widthScale = toUniformScaleX / toUniformScaleY;

30 projectionMatrix[0] \*= widthScale;

projectionMatrix[8] \*= widthScale;

**return** widthScale;

}**return** 1f;

35 }**public static float** transformProjectionFitSmallestSide(**float**[] projectionMatrix, *,→* Pair<Integer, Integer> screenResolution, Pair<Integer, Integer> *,→*

neuralFrameResolution) {

**if** (mCurrentAdjustment == AdjustmentType.FrustumCrop || mCurrentAdjustment *,→* == AdjustmentType.FrustumCropZoom) {

**float** toUniformScaleX = screenResolution.first / *,→*

(**float**)neuralFrameResolution.first;

**float** toUniformScaleY = screenResolution.second / *,→*

(**float**)neuralFrameResolution.second;

40 **if** (toUniformScaleY > toUniformScaleX) {

**float** heightScaler = toUniformScaleY / toUniformScaleX;

projectionMatrix[5] \*= heightScaler;

projectionMatrix[9] \*= heightScaler;

**return** heightScaler;

45 } **else** {

**float** widthScaler = toUniformScaleX / toUniformScaleY;

projectionMatrix[0] \*= widthScaler;

projectionMatrix[8] \*= widthScaler;

50 }

**return** -widthScaler;

}

71

}**return** 1f;

**public static void** transformProjectionFitZoomedWidth(**float**[] projectionMatrix, *,→* Pair<Integer, Integer> screenResolution, Pair<Integer, Integer> *,→*

neuralFrameResolution, **float** zoomFactor) {

55 **if** (mCurrentAdjustment == AdjustmentType.UvMapToZoomedWidth) { **float** toUniformScaleX = screenResolution.first / *,→*

(**float**)neuralFrameResolution.first;

**float** toUniformScaleY = screenResolution.second / *,→*

(**float**)neuralFrameResolution.second;

**float** commonScale = toUniformScaleX \* zoomFactor;

**float** widthScale = toUniformScaleX / commonScale;

60 **float** heightScale = toUniformScaleY / commonScale;

projectionMatrix[0] \*= widthScale;

projectionMatrix[8] \*= widthScale;

projectionMatrix[5] \*= heightScale;

65 }

projectionMatrix[9] \*= heightScale;

}**public static void** transformProjectionFlipVertically(**float**[] projectionMatrix, *,→* **boolean** flipVertically) {

**if** (flipVertically) { projectionMatrix[5] \*= -1f; } }

70 **public static boolean** transformProjectionUnrotatedCrop(**float**[] projectionMatrix, *,→* ProjectionCropper cropper, **float**[] cropCenterInView, **float** *,→*

cropHeightInWorld, **float** unstretchingScaler) {

**if** (mCurrentAdjustment == AdjustmentType.FrustumCrop || mCurrentAdjustment *,→* == AdjustmentType.FrustumCropZoom) {

cropper.cropProjection(projectionMatrix, cropCenterInView, *,→*

cropHeightInWorld, unstretchingScaler);

}

**return true**;

75 **return false**;

}**public static** AdjustmentType getCurrentAdjustment() { **return** mCurrentAdjustment; } }

Listing A.6: SmplxInferer.java

Comment: The SmplxInferer implements the full algorithm of SMPL-X inference, except for blending of pose blendshapes, since they would be too costly to calculate on every frame.

**public class** SmplxInferer {

**public class** GlobalTranslation {

**private float**[] fullPose;

**public** GlobalTranslation(**float**[] fullPose) {

5 **this**.fullPose = fullPose;

}**public float** getX() { **return** fullPose[0]; }

**public float** getY() { **return** fullPose[1]; }

**public float** getZ() { **return** fullPose[2]; }

10 **public float**[] getUnsafeArray() { **return** fullPose; }

}**private float**[] mJointsPositions, mTemplateVertices, mShapedVertices, *,→* mShapedirs, mWeights, mJointsRegressor;

**private float**[] mHandsMeanL, mHandsMeanR;

**private int**[] mParentsMap;

15 **private float**[] mFlatPose;

**private float**[][] mUncorrectedJointsTranslations = **new** *,→*

**float**[N\_JOINTS][VERTEX\_ELEMENTS];

**private** List<**float**[]> mViewSpaceJointsTranslations = **new** ArrayList<>(); **private** FloatBuffer mRotationMatrices, mJointsTransforms;

**private int** mTotalVertices;

20 **private float** mMaxY, mMinY, mHeelsY;

**public static final int**[] BBOX\_JOINTS = **new int**[] {

0, // pelvis

4,10, // heel, knee

25 5,11, // heel, knee

18, 20, // palm, elbow

19, 21, // palm, elbow

};

23 // head top

30 **public static final int** LIFT\_JOINT = 23;

**public static final float**[] LIFT\_AMOUNT = {0.0f, 0.3f, 0.0f}; **public static final int** BETAS\_ELEMENTS = 10, EXPRESSION\_ELEMENTS = 10;

72

**public static final int** N\_JOINTS = 55, POSE\_ENTRY\_ELEMENTS = 3;

**private static final int** VERTEX\_ELEMENTS = 3, HOMOGENEOUS\_ELEMENTS = 4; 35 **private static final int** MATRIX\_SIZE = HOMOGENEOUS\_ELEMENTS \* HOMOGENEOUS\_ELEMENTS; **private static final int**

GLOBAL\_TRANSLATION\_ELEMENTS = 1 \* POSE\_ENTRY\_ELEMENTS,

POSE\_GLOBAL\_POSE\_ELEMENTS = 1 \* POSE\_ENTRY\_ELEMENTS,

POSE\_BODY\_POSE\_ELEMENTS = 21 \* POSE\_ENTRY\_ELEMENTS,

40 POSE\_JAW\_ELEMENTS = 1 \* POSE\_ENTRY\_ELEMENTS,

POSE\_EYES\_ELEMENTS = 2 \* POSE\_ENTRY\_ELEMENTS,

POSE\_LEFT\_HAND\_POSE\_ELEMENTS = 15 \* POSE\_ENTRY\_ELEMENTS,

POSE\_RIGHT\_HAND\_POSE\_ELEMENTS = 15 \* POSE\_ENTRY\_ELEMENTS;

**private static final int**

45 GLOBAL\_TRANSLATION\_OFFSET = 0,

POSE\_GLOBAL\_POSE\_OFFSET = GLOBAL\_TRANSLATION\_ELEMENTS,

POSE\_BODY\_POSE\_OFFSET = POSE\_GLOBAL\_POSE\_OFFSET + *,→*

POSE\_GLOBAL\_POSE\_ELEMENTS,

POSE\_JAW\_OFFSET = POSE\_BODY\_POSE\_OFFSET + POSE\_BODY\_POSE\_ELEMENTS,

POSE\_EYES\_OFFSET = POSE\_JAW\_OFFSET + POSE\_JAW\_ELEMENTS,

50 POSE\_LEFT\_HAND\_POSE\_OFFSET = POSE\_EYES\_OFFSET + POSE\_EYES\_ELEMENTS, POSE\_RIGHT\_HAND\_POSE\_OFFSET = POSE\_LEFT\_HAND\_POSE\_OFFSET + *,→*

POSE\_LEFT\_HAND\_POSE\_ELEMENTS;

**private static final int** MAT4\_TRANSLATION\_OFFSET = HOMOGENEOUS\_ELEMENTS \* *,→* (HOMOGENEOUS\_ELEMENTS - 1);

**private static final float**[] TEMPORARY\_MAT = **new float**[HOMOGENEOUS\_ELEMENTS \* *,→* HOMOGENEOUS\_ELEMENTS];

**private static final float**[] TEMPORARY\_VEC = **new float**[HOMOGENEOUS\_ELEMENTS]; 55 **public static final int** FRAME\_SIZE = N\_JOINTS \*POSE\_ENTRY\_ELEMENTS + *,→* GLOBAL\_TRANSLATION\_ELEMENTS;

**public** SmplxInferer(String npzAbsolutePath, **float**[] betas, **float**[] expression) { NpzFile.Reader npz = NpzFile.read(Paths.get(npzAbsolutePath));

**int** readBufferSizeBytes = 1 << 18;

60 mTemplateVertices = npz.get("v\_template", *,→*

readBufferSizeBytes).asFloatArray();

mShapedirs = npz.get("shapedirs", readBufferSizeBytes).asFloatArray(); mJointsRegressor = npz.get("J\_regressor", readBufferSizeBytes).asFloatArray(); mWeights = npz.get("weights", readBufferSizeBytes).asFloatArray(); mParentsMap = npz.get("kintree\_table", 512).asIntArray();

65 mParentsMap[0] = -1;

mHandsMeanL = npz.get("hands\_meanl", 512).asFloatArray();

mHandsMeanR = npz.get("hands\_meanr", 512).asFloatArray();

npz.close();

mFlatPose = **new float**[FRAME\_SIZE];

70 Arrays.fill(mFlatPose, 0f);

recalculateShape(betas, expression);

**for**(**int** i = 0; i < BBOX\_JOINTS.length; ++i) {

}

75 }

mViewSpaceJointsTranslations.add(**new float**[VERTEX\_ELEMENTS]);

**public void** recalculateShape(**float**[] betas, **float**[] expression) {

**float**[] shapeComponents = **new float**[BETAS\_ELEMENTS + EXPRESSION\_ELEMENTS]; System.arraycopy(betas, 0, shapeComponents, 0, BETAS\_ELEMENTS);

80 System.arraycopy(expression, 0, shapeComponents, BETAS\_ELEMENTS, *,→* EXPRESSION\_ELEMENTS);

mShapedVertices = Arrays.copyOf(mTemplateVertices, mTemplateVertices.length); mTotalVertices = mShapedVertices.length / VERTEX\_ELEMENTS;

addDotOverLastDim(mShapedirs, shapeComponents, /\*destination\*/mShapedVertices); mJointsPositions = **new float**[N\_JOINTS \* VERTEX\_ELEMENTS];

85 Arrays.fill(mJointsPositions, 0f);

matMulMat(mJointsRegressor, mShapedVertices, mTotalVertices, *,→*

/\*destination\*/mJointsPositions);

computeBodyBounds(mShapedVertices, mWeights);

**float**[] zeros = **new float**[N\_JOINTS \* HOMOGENEOUS\_ELEMENTS \* *,→*

HOMOGENEOUS\_ELEMENTS];

Arrays.fill(zeros, 0f);

90 **final int** MATRIX\_SIZE = HOMOGENEOUS\_ELEMENTS \* HOMOGENEOUS\_ELEMENTS; mRotationMatrices = ByteBuffer.allocateDirect(N\_JOINTS \* MATRIX\_SIZE \* *,→* Float.BYTES)

.order(ByteOrder.nativeOrder())

.asFloatBuffer();

**for** (**int** jointIdx = 0; jointIdx < N\_JOINTS; ++jointIdx) {

95 **for**(**int** coord = 0; coord < VERTEX\_ELEMENTS; ++coord) { **float** jointCoordVal = mJointsPositions[jointIdx \* VERTEX\_ELEMENTS + *,→* coord];

73

**float** parentCoordVal = (jointIdx != 0) ? *,→*

mJointsPositions[mParentsMap[jointIdx] \* VERTEX\_ELEMENTS + coord] *,→* : 0f;

zeros[jointIdx\*MATRIX\_SIZE + 12 + coord] = jointCoordVal - *,→* parentCoordVal;

}

100 zeros[jointIdx\*MATRIX\_SIZE + 15] = 1f;

}mRotationMatrices.put(zeros);

mRotationMatrices.rewind();

Arrays.fill(zeros, 0f);

105 mJointsTransforms = ByteBuffer.allocateDirect(N\_JOINTS \* MATRIX\_SIZE \* *,→* Float.BYTES)

.order(ByteOrder.nativeOrder())

.asFloatBuffer();

mJointsTransforms.put(zeros);

110 }

mJointsTransforms.rewind();

**public void** computeBodyBounds(**float**[] shapedVertices, **float**[] *,→* vertexWeightsPerJoint) {

mMinY = Float.MAX\_VALUE;

mMaxY = Float.MIN\_VALUE;

115 **int** weightIdx = 0;

mHeelsY = 0f;

**int** heelVerticesCount = 0;

**for** (**int** i = 1; i < shapedVertices.length; i+=3) {

**float** y = shapedVertices[i];

120 **if** (y < mMinY) {

mMinY = y;

}**if** (y > mMaxY) {

mMaxY = y;

125 }**if** (vertexWeightsPerJoint[weightIdx + 7] > 0.05f || *,→* vertexWeightsPerJoint[weightIdx + 8] > 0.05f || *,→*

vertexWeightsPerJoint[weightIdx + 10] > 0.05f || *,→*

vertexWeightsPerJoint[weightIdx + 11] > 0.05f) {

mHeelsY += y;

++heelVerticesCount; }

130 weightIdx += 55;

}mHeelsY /= heelVerticesCount;

Logging.d("SMPLX " + mMinY + " " + mMaxY + " " + mHeelsY + " " + *,→*

heelVerticesCount + " " + mJointsPositions[0\*VERTEX\_ELEMENTS + 1] ); }

135 // matrix "a" of shape [aRows, commonDimension]

// matrix "b" of shape [commonDimension, bCols]

// matrix "destination" of shape [aRows, bCols]

**public static void** matMulMat(**float**[] a, **float**[] b, **int** commonDimension, **float**[] *,→* destination) {

**int** bCols = b.length / commonDimension;

140 **int** aOffset = 0, destinationOffset = 0;

**while**(destinationOffset < destination.length) {

**int** bIdx = 0;

**for**(**int** aIdx = aOffset; aIdx < aOffset + commonDimension; ++aIdx) {

**for**(**int** destinationIdx = destinationOffset; destinationIdx < *,→*

destinationOffset + bCols; ++destinationIdx) {

145 destination[destinationIdx] += a[aIdx] \* b[bIdx];

}

++bIdx;

}aOffset += commonDimension;

150 destinationOffset += bCols;

}

}**public static void** matMulMatSquare(**int** dim, FloatBuffer a, **int** aOffset, *,→* FloatBuffer b, **int** bOffset, FloatBuffer destination) {

Arrays.fill(TEMPORARY\_MAT, 0f);

155 **int** destinationOffset = 0;

**while**(destinationOffset < TEMPORARY\_MAT.length) {

**int** bIdx = bOffset;

**for**(**int** aIdx = aOffset; aIdx < aOffset + dim; ++aIdx) {

74

**for**(**int** destinationIdx = destinationOffset; destinationIdx < *,→* destinationOffset + dim; ++destinationIdx){

160 TEMPORARY\_MAT[destinationIdx] += a.get(aIdx) \* b.get(bIdx);

}

++bIdx;

}aOffset += dim;

165 destinationOffset += dim;

}destination.put(TEMPORARY\_MAT);

}// tensor of shape "[a, b, c, ..., x, y, z]"

170 // vector of length "z"

// destination shape "[a, b, c, ..., x, y]"

// each dot product is ADDED to what currently sits in "destination" **public static void** addDotOverLastDim(**float**[] tensor, **float**[] vector, **float**[] *,→* destination) {

**int** offset = 0;

175 **int** totalDots = tensor.length / vector.length;

**for** (**int** dotIdx = 0; dotIdx < totalDots; ++dotIdx) {

**for** (**int** z = 0; z < vector.length; ++z) {

destination[dotIdx] += tensor[offset + z] \* vector[z]; }

180 offset += vector.length;

}

}// transforms: 55 x 4 x 4

// worldJointsPositions: 55 x 3

185 **private static void** correctJointTranslation(FloatBuffer transforms, **float**[] *,→* worldJointsPositions) {

**int** matOffset = 0, jointOffset = 0;

**for** (**int** jointIdx = 0; jointIdx < N\_JOINTS; ++jointIdx) {

**int** matInnerOffset = 0;

transforms.position(matOffset + MAT4\_TRANSLATION\_OFFSET);

190 transforms.get(TEMPORARY\_VEC);

**for** (**int** coord = 0; coord < VERTEX\_ELEMENTS; ++coord) {

**for** (**int** transformColumn = 0; transformColumn < VERTEX\_ELEMENTS; *,→* ++transformColumn) {

TEMPORARY\_VEC[transformColumn] -= *,→*

worldJointsPositions[jointOffset + coord]

\* transforms.get(matOffset + matInnerOffset);

195 ++matInnerOffset;

}++matInnerOffset;

}transforms.position(matOffset + MAT4\_TRANSLATION\_OFFSET);

200 transforms.put(TEMPORARY\_VEC);

matOffset += MATRIX\_SIZE;

}

}

jointOffset += VERTEX\_ELEMENTS;

205 **public void** inferJointsTransforms() {

mRotationMatrices.rewind();

computeRodriguesRotationMatrices(mFlatPose, /\*pose offset\*/ *,→*

POSE\_GLOBAL\_POSE\_OFFSET, /\*destination\*/ mRotationMatrices);

mRotationMatrices.rewind();

210 mJointsTransforms.rewind();

computeJointsTransforms(mRotationMatrices, mParentsMap, /\*destination\*/ *,→* mJointsTransforms);

mJointsTransforms.rewind();

**for** (**int** j = 0; j < N\_JOINTS; ++j) {

**for** (**int** coord = 0; coord < VERTEX\_ELEMENTS; ++coord) {

215 mUncorrectedJointsTranslations[j][coord] = *,→*

mJointsTransforms.get(j\*MATRIX\_SIZE + MAT4\_TRANSLATION\_OFFSET + *,→*

}

}

coord);

correctJointTranslation(mJointsTransforms, mJointsPositions); 220 mJointsTransforms.rewind();

}

75

**private void** computeRodriguesRotationMatrices(**float**[] pose, **int** poseOffset, *,→* FloatBuffer destination4x4) {

**int** matrixOffset = 0;

225 **for**(**int** entryOffset = poseOffset; entryOffset < pose.length; *,→* entryOffset+=VERTEX\_ELEMENTS) {

**double** angle = Math.sqrt(pose[entryOffset]\*pose[entryOffset]

+ pose[entryOffset+1]\*pose[entryOffset+1]

+ pose[entryOffset+2]\*pose[entryOffset+2]);

**float** invNorm = (**float**)(1.0/(angle + 1e-8));

230 **float** sin = (**float**)Math.sin(angle);

**float** cosSub1 = 1f - (**float**)Math.cos(angle);

**float** x = pose[entryOffset ] \* invNorm;

**float** y = pose[entryOffset + 1] \* invNorm;

**float** z = pose[entryOffset + 2] \* invNorm;

235 **float** cxy = cosSub1\*x\*y, cxz = cosSub1\*x\*z, cyz = cosSub1\*y\*z; **float** \_cxx = -cosSub1\*x\*x, \_cyy = -cosSub1\*y\*y, \_czz = -cosSub1\*z\*z;

destination4x4.position(matrixOffset);

destination4x4.put(1f + \_cyy + \_czz);

240 destination4x4.put(z\*sin + cxy);

destination4x4.put(-y\*sin + cxz);

destination4x4.put(0f);

destination4x4.put(-z\*sin + cxy);

destination4x4.put(1f + \_cxx + \_czz)

245 destination4x4.put(x\*sin + cyz);

destination4x4.put(0f);

destination4x4.put(y\*sin + cxz);

destination4x4.put(-x\*sin + cyz);

destination4x4.put(1f + \_cxx + \_cyy);

250 matrixOffset += MATRIX\_SIZE;

}

}

**private void** computeJointsTransforms(FloatBuffer rotationMatrices, **int**[] *,→* parentByChildMap, FloatBuffer destination) {

255 // put root transform unchanged

rotationMatrices.limit(MATRIX\_SIZE);

destination.put(rotationMatrices);

rotationMatrices.limit(rotationMatrices.capacity());

// compute relative transforms for each child joint

260 **int** currentOffset = MATRIX\_SIZE;

**for**(**int** child = 1; child < parentByChildMap.length; ++child) {

matMulMatSquare(HOMOGENEOUS\_ELEMENTS,

rotationMatrices, currentOffset,

destination, parentByChildMap[child] \* MATRIX\_SIZE,

265 destination);

}

}

currentOffset += MATRIX\_SIZE;

270 **public void** fillWholePose(**float**[] wholePose) {

**if** (wholePose != **null** && wholePose.length == mFlatPose.length) {

}

}

mFlatPose = wholePose;

275**public** GlobalTranslation getGlobalTranslation() {**return new** *,→* GlobalTranslation(mFlatPose);}

**public** Mesh constructOpenGlMesh(OpenGlRender render, String uvAssetPath, String *,→* facesAssetPath) **throws** IOException {

**final int** VECTOR\_SIZE = 4;

280 **final int** POSITION\_COORDS = 3;

**final int** POSITION\_BYTES = POSITION\_COORDS\*Float.BYTES;

**final int** UV\_COORDS = 2;

**final int** UV\_BYTES = UV\_COORDS\*Float.BYTES;

**final int** N\_BIGGEST\_WEIGHTS\_VBOS = 12 / 4;

285 **final int** WEIGHTS\_BYTES = Float.BYTES \* VECTOR\_SIZE;

**final int** WEIGHTS\_INDICES\_BYTES = Integer.BYTES \* VECTOR\_SIZE;

**final int** STRIDE = POSITION\_BYTES + UV\_BYTES + *,→*

N\_BIGGEST\_WEIGHTS\_VBOS\*WEIGHTS\_BYTES + *,→*

N\_BIGGEST\_WEIGHTS\_VBOS\*WEIGHTS\_INDICES\_BYTES;

ByteBuffer allVertexAttributes = ByteBuffer.allocateDirect(STRIDE \* *,→* mTotalVertices)

.order(ByteOrder.nativeOrder());

76

290IndexBuffer indexBuffer = Mesh.parseNumpyF(render, facesAssetPath, 3);

VertexBufferMeta[] allVbosMeta = **new** VertexBufferMeta[1 + 1 + *,→*

N\_BIGGEST\_WEIGHTS\_VBOS + N\_BIGGEST\_WEIGHTS\_VBOS];

allVbosMeta[0] = **new** VertexBufferMeta(GLES30.GL\_FLOAT, 3, **false**, STRIDE, 0); 295 FloatBuffer textureCoords = Mesh.parseNumpyTextureCoords(render, *,→* uvAssetPath, 2);

allVbosMeta[1] = **new** VertexBufferMeta(GLES30.GL\_FLOAT, 2, **false**, STRIDE, *,→* POSITION\_BYTES);

**for** (**int** i = 0; i < N\_BIGGEST\_WEIGHTS\_VBOS; ++i){

**int** weightsOffset = (POSITION\_BYTES + UV\_BYTES) + (WEIGHTS\_BYTES + *,→* WEIGHTS\_INDICES\_BYTES)\*i;

**int** indicesOffset = weightsOffset + WEIGHTS\_BYTES;

300 allVbosMeta[2+i\*2] = **new** VertexBufferMeta(GLES30.GL\_FLOAT, 4, **false**, *,→* STRIDE, weightsOffset);

allVbosMeta[2+i\*2+1] = **new** VertexBufferMeta(GLES30.GL\_INT, 4, **false**, *,→*

STRIDE, indicesOffset);

}

**for** (**int** v = 0; v < mTotalVertices; ++v) {

305 // position

allVertexAttributes.putFloat(mShapedVertices[3\*v]);

allVertexAttributes.putFloat(mShapedVertices[3\*v+1]);

allVertexAttributes.putFloat(mShapedVertices[3\*v+2]);

// uvs

310 allVertexAttributes.putFloat(textureCoords.get(2\*v)); allVertexAttributes.putFloat(textureCoords.get(2\*v+1));

// smplx weights and indices of biggest transform matrices

**long**[] hackySortedIndices = hackySortIndices(mWeights, v\* N\_JOINTS, *,→* N\_JOINTS);

**for**(**int** wVbo = 0; wVbo < N\_BIGGEST\_WEIGHTS\_VBOS; ++wVbo) {

315 **for**(**int** coord = 0; coord < 4; ++coord){

allVertexAttributes.putFloat(getHackyValue(hackySortedIndices, *,→*

N\_JOINTS - 1 - wVbo\*4 - coord));

}**for**(**int** coord = 0; coord < 4; ++coord){

allVertexAttributes.putInt(getHackyIndex(hackySortedIndices, *,→*

320 }

N\_JOINTS - 1 - wVbo\*4 - coord));

}

}

VertexBuffer interleavedVbo = **new** VertexBuffer(render, *,→*

3+2+N\_BIGGEST\_WEIGHTS\_VBOS\*8, GlConstants.BufferHint.STATIC\_DRAW, *,→*

allVertexAttributes, 1);

325 **return new** Mesh(render, Mesh.PrimitiveMode.TRIANGLES, indexBuffer, *,→*

interleavedVbo, allVbosMeta);

}

**public** FloatBuffer getJointsTransforms() {

**return** mJointsTransforms;

330 }**public float** getCenterToHeelsOffsetY() { **return** -mHeelsY; } **public float** getTPoseHeight() { **return** mMaxY - mHeelsY; }

**public** List<**float**[]> computeViewDimensions(**float**[] modelViewMatrix, **float**[] *,→* outputMin, **float**[] outputMax) {

335 **for** (**int** viewCoord = 0; viewCoord < VERTEX\_ELEMENTS; ++viewCoord) { outputMin[viewCoord] = Float.POSITIVE\_INFINITY;

outputMax[viewCoord] = Float.NEGATIVE\_INFINITY;

mUncorrectedJointsTranslations[LIFT\_JOINT][viewCoord] += *,→*

LIFT\_AMOUNT[viewCoord];

}

340**for** (**int** jointIdx = 0; jointIdx < BBOX\_JOINTS.length; ++jointIdx) { RenderingHelper.matMulPoint(modelViewMatrix, *,→*

mUncorrectedJointsTranslations[BBOX\_JOINTS[jointIdx]], *,→*

TEMPORARY\_VEC);

**for** (**int** viewCoord = 0; viewCoord < VERTEX\_ELEMENTS; ++viewCoord) { 345 outputMin[viewCoord] = Math.min(outputMin[viewCoord], *,→* TEMPORARY\_VEC[viewCoord]);

77

}

}

outputMax[viewCoord] = Math.max(outputMax[viewCoord], *,→* TEMPORARY\_VEC[viewCoord]);

mViewSpaceJointsTranslations.get(jointIdx)[viewCoord] = *,→* TEMPORARY\_VEC[viewCoord];

350 **return** mViewSpaceJointsTranslations;

}

**private long**[] hackySortIndices(**float**[] buffer, **int** offset, **int** length) { **long**[] valueKeyPairs = **new long**[length];

355 **for** (**int** j = 0; j < length; ++j) {

valueKeyPairs[j] = (((**long**) Float.floatToIntBits(buffer[offset + j])) << *,→* 32) | (j & 0xffffffffL);

}Arrays.sort(valueKeyPairs);

**return** valueKeyPairs;

360 }**private int** getHackyIndex(**long**[] hackyArray, **int** idx) {**return** (**int**) *,→* (hackyArray[idx]);}

**private float** getHackyValue(**long**[] hackyArray, **int** idx) {**return** *,→*

Float.intBitsToFloat((**int**) (hackyArray[idx] >> 32));}

}

Listing A.7: neural\_render\_smplx.vert

Comment: This is the main Vertex shader used for posing of every vertex in a body mesh from a default pose into a certain pose, given 55 matrices of joints rotations. For each vertex, the shader receives indices of the most significant joints, and finds a weighted sum of matrices accordingly. Then, each vertex is projected from the local body coordinate space, into the world space, into the

camera space, and into the image coordinate space. This all is done with a single 4 *×* 4 matrix that merges the 3 transforms.

**#version** 300 es

**#define** N\_SMPLX\_JOINTS 55

**precision highp** float;

**uniform mat4** u\_ModelViewProjection;

5 **uniform mat4** u\_JointsTransforms[N\_SMPLX\_JOINTS];

**layout**(**location** = 0) **in vec4** a\_TemplatePosition;

**layout**(**location** = 1) **in vec2** a\_TexCoord;

**layout**(**location** = 2) **in vec4** a\_BiggestJointWeights\_0;

10 **layout**(**location** = 3) **in ivec4** a\_BiggestJointWeightsIdx\_0;

**layout**(**location** = 4) **in vec4** a\_BiggestJointWeights\_1;

**layout**(**location** = 5) **in ivec4** a\_BiggestJointWeightsIdx\_1;

**layout**(**location** = 6) **in vec4** a\_BiggestJointWeights\_2;

**layout**(**location** = 7) **in ivec4** a\_BiggestJointWeightsIdx\_2;

15**out vec2** v\_TexCoord;

void main() {

// NOTE:

20 // 4 first lines still give acceptable quality of SMPLX

// 10 first lines is the maximum number of non-zero weights for all mesh vertices // 12 lines are here is just to be safe

**mat4** uniqueVertexTransform =

(

25 (u\_JointsTransforms[a\_BiggestJointWeightsIdx\_0.x] \* a\_BiggestJointWeights\_0.x + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_0.y] \* a\_BiggestJointWeights\_0.y )

+ (

30 u\_JointsTransforms[a\_BiggestJointWeightsIdx\_0.z] \* a\_BiggestJointWeights\_0.z + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_0.w] \* a\_BiggestJointWeights\_0.w

)+

)

35 ((u\_JointsTransforms[a\_BiggestJointWeightsIdx\_1.x] \* a\_BiggestJointWeights\_1.x + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_1.y] \* a\_BiggestJointWeights\_1.y )

78

40 + (u\_JointsTransforms[a\_BiggestJointWeightsIdx\_1.z] \* a\_BiggestJointWeights\_1.z + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_1.w] \* a\_BiggestJointWeights\_1.w

)

45 +

)

((u\_JointsTransforms[a\_BiggestJointWeightsIdx\_2.x] \* a\_BiggestJointWeights\_2.x + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_2.y] \* a\_BiggestJointWeights\_2.y 50 )

// + (

// u\_JointsTransforms[a\_BiggestJointWeightsIdx\_2.z] \* *,→*

a\_BiggestJointWeights\_2.z

// + u\_JointsTransforms[a\_BiggestJointWeightsIdx\_2.w] \* *,→*

// )

a\_BiggestJointWeights\_2.w

55 );uniqueVertexTransform[3][3] = 1.0;

gl\_Position = u\_ModelViewProjection \* (uniqueVertexTransform \* a\_TemplatePosition);

}

Listing A.8: neural\_render.frag

v\_TexCoord = a\_TexCoord;

Comment: This is the main Fragment shader used for rasterization of the input frame for the neural network. For each pixel, is uses interpolated texture coordinates that allow to sample a certain point at the neural texture. We use multiple FP16 4-channel slices to construct the *N*-channel neural texture. The shader samples the slices in a loop, then quantizes the floating-point numbers into INT8 and using bit-shift packs the values into the output frame, which can store up to 16 bytes of data per pixel (4 channels, 32-bit integer number each).

**#version** 300 es

**precision highp** float;

**#ifndef** TRY\_LEGACY

**precision highp uint**;

5 **#else**

**precision highp** int;

**#endif**

**precision highp sampler2DArray**;

**#if** !defined(QUANTIZATION) || !defined(N\_COMPONENT\_BYTES) || !defined(N\_COMPONENTS) 10 **#error**

**#endif**

**uniform sampler2DArray** u\_NeuralTexture;

**uniform** int u\_NeuralQuantizationOfZero;

15 **uniform** float u\_ReciprocalNeuralQuantizationDelta;

**#ifdef** RENDER\_UV

**uniform** int u\_UvQuantizationOfZero;

**uniform** float u\_ReciprocalUvQuantizationDelta;

**#endif**

20**in vec2** v\_TexCoord;

**layout**(**location** = 0) **out uvec4** o\_PackedNeuralFrag;

**#ifdef** RENDER\_UV

25 **layout**(**location** = 1) **out uvec2** o\_UvFrag;

**#endif**

**uvec4** getNeuralPixelSlice(int index) {

return **uvec4**(**ivec4**(texture(u\_NeuralTexture, **vec3**(v\_TexCoord, index)) \* *,→*

30 }

void main() {

u\_ReciprocalNeuralQuantizationDelta) + u\_NeuralQuantizationOfZero);

**uvec4** packingMultiplier = **uvec4**(1U, 256U, 65536U, 16777216U);

**uvec4** channelOffsets = **uvec4**(0U, (N\_COMPONENT\_BYTES), (N\_COMPONENT\_BYTES\*2U), *,→* (N\_COMPONENT\_BYTES\*3U));

35 **uvec4** rgbaOffsets = channelOffsets % 4U;

**uvec4** sliceOffsets = channelOffsets / 4U;

**uvec4** quantizedSlices[N\_SLICES];

for(int i = 0; i < N\_SLICES; ++i) { quantizedSlices[i] = getNeuralPixelSlice(i); } 79