Realtime 3D Human Mesh Reconstruction from a single 2D Camera View

Sophie Kernchen, Dawid Wlodarczak, Alexander Ehrenhöfer

TU Berlin

***Abstract*— Computer Vision has experienced remarkable advances in recent years, with both traditional and especially machine learning-based methods showing notable promise. Classic approaches, guided by carefully crafted heuristics and assumptions, run efficiently but tend to introduce artifacts that stem from these domain-specific priors. Meanwhile, deep learning methods generate highly detailed and visually convincing results, yet often require substantial computation and processing time. In this work, we explore machine learning methods and try to construct a real-time solution for a mesh reconstruction task. We construct a purely visual-based application from 2D image input for real-time 3D human mesh reconstruction. The solution will be presented through a pipeline that takes a webcam image and streams a coarsely constructed mesh to Unity.**

I. Introduction

As part of the Advanced Web Technologies student project led by the Fraunhofer Institute for Open Communication Systems[1], we are investigating a monocular approach for real-time human reconstruction in Unity. We begin by going through relevant research that also focuses on human mesh reconstruction from image, where we briefly mention the methodology that was used. Next, we briefly describe our setup and present our solution in the form of a pipeline. We explain each component and give an insight into our design choices. In the end, we discuss our solution and give a short conclusion.

In contrast to highly sophisticated end-to-end systems, we will present a modular approach, which we developed from scratch. We designed a solution that includes several basic key elements and, over the course of the project, improved each of these elements as much as possible in terms of time and quality.

The source code and further demo material is available on our github repository[2].

Performance analyses and time measurements shown in this paper are taken from tests on a Lenovo Thinkpad X1 with a GeForce 1050, so a typical private laptop was used and not a high performance GPU machine.

[1] fokus.fraunhofer.de

[2] github.com/aalolexx/awt-pj-ws2425-3d-realtime-reconstruction

II. Related Work

Accurately estimating the runtime of the following methods is challenging, as performance always depends on factors such as image resolution, hardware specifications, and system optimizations. Even when runtimes are listed by the authors, it remains difficult to predict how quickly a given program would run on our specific setup without implementing each approach ourselves. Consequently, the papers mentioned here are intended only as a rough comparison for our own implementation. According to our research, certain models can operate within a few milliseconds, while others may require up to several seconds, which serves as a general benchmark for our evaluation.

HMR [1] is an end-to-end framework that directly regresses the parameters of a parametric human body model (SMPL) from a single image. By combining deep convolutional networks with a differentiable re-projection loss, it estimates both 3D pose and shape from 2D input without needing intermediate key point detection.

Tex2Shape [2] refines a parametric body model from a single image by learning displacements that capture high-frequency surface details, enabling more realistic reconstructions of clothing and body features.

DeepHuman [3] adopts a volumetric approach for single-image human reconstruction via a coarse-to-fine strategy. It first infers a coarse occupancy volume and then refines details using learned features, resulting in more accurate volumetric meshes.

PIFu [4] learns a continuous implicit representation of the human surface that is "pixel-aligned" with image features. Given one or more images, it predicts whether a 3D point in space is inside or outside the surface, enabling high-fidelity reconstructions.

PIFuHD [5] extends PIFu to higher-resolution reconstructions by incorporating multi-level pixel-aligned features. This captures finer details in clothing, hair, and face, producing more realistic and detailed 3D meshes from single images.

III. The Pipeline

We present a fully modular pipeline that processes a single 2D camera stream of a human and transforms it into a real-time 3D representation suitable for visualization in Unity

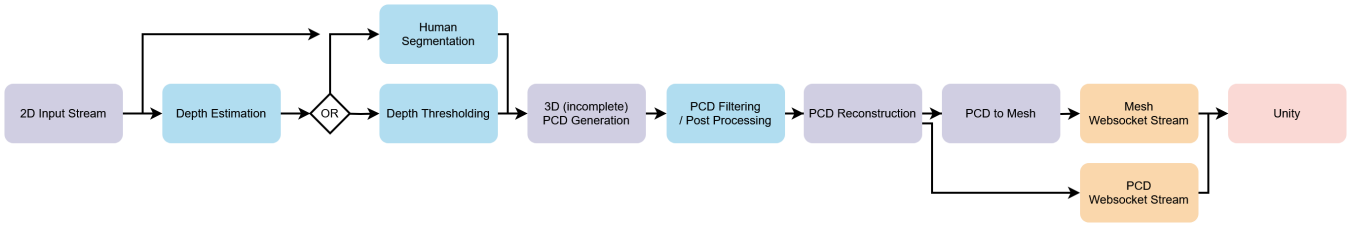

Fig. 1. Overview of the 3D reconstruction pipeline.

(see Fig. 1). This system operates in Python and follows a module-based architecture, where each module functions as an independent component, taking an input, processing it, and returning an output. The pipeline is initiated via the command line and includes additional visualization options beyond Unity. Additionally, the system tracks processing times for each module to evaluate overall performance.

A. Pipeline Components

2D Input Stream
A standard 2D video input from a webcam or monocular camera. No special background preparation required.

Depth Estimation
We use an existing machine learning model to predict a depth map for each frame, approximating the 3D scene structure.

Foreground Segmentation
A segmentation model extracts a mask identifying the human subject and filtering out background noise.

PCD Generation
An initial 3D point cloud (PCD) is constructed using the depth estimation and foreground mask, representing only the visible portion of the human.

PCD Filtering and Post-Processing
Noise reduction techniques refine the PCD by removing outliers and artifacts.

PCD Reconstruction
Our own proposed CNN model reconstructs a complete human body representation from the incomplete PCD.

PCD to Mesh Conversion
The reconstructed point cloud is then converted into a 3D mesh using standard meshing algorithms.

Streaming and Unity Visualization
The final mesh is streamed to Unity in real-time for rendering and interaction, with additional visualization options available.

IV. Point Cloud Estimation

A key step in the 3D reconstruction pipeline is the generation of the point cloud from the processed 2D input. This process consists of three main stages: depth estimation, foreground segmentation, and the final point cloud construction.

Depth Estimation
For depth estimation, we employ the DepthAnythingV2 [6] model, which provides high-quality depth predictions from a single RGB frame. To achieve an optimal balance between accuracy and processing time, we conducted extensive performance evaluations (see Fig. 2). On our test laptop, the depth estimation step takes approximately 100 ms per frame, making it feasible for near real-time applications. Although histogram equalization was tested as a preprocessing step to enhance depth details, it did not yield significant improvements. Consequently, the raw depth map output from the model is used directly.

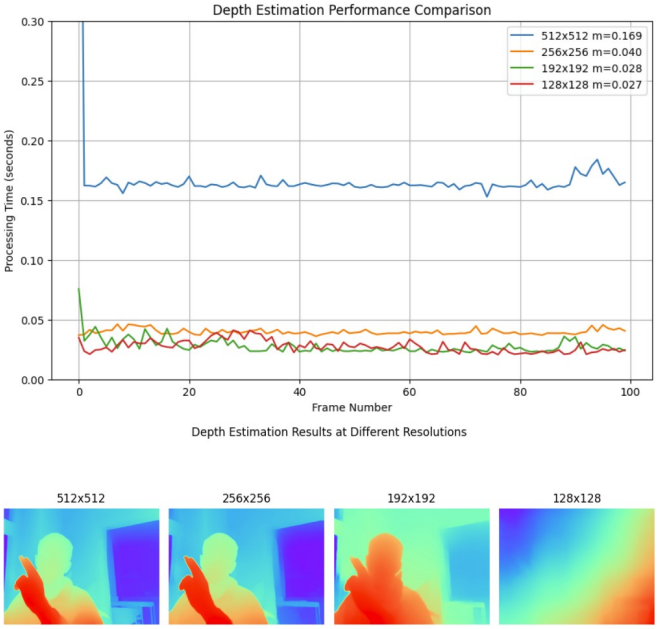

Fig. 2. Performance Analysis of the DepthAnythingV2 Model.

Foreground Segmentation
Foreground segmentation is crucial to isolating the human subject from the background in the depth map or original frame. Several approaches were evaluated for this task. The Segment Anything Model (SAM) [7] and the RMBG model [8] demonstrated excellent segmentation quality, but their computational cost was too high for real-time processing. To improve efficiency, we explored classical image processing techniques, including a contour-based approach (see Fig. 3)

followed by a gradient/edge-based method (see Fig. 4). While the gradient-based approach proved robust, we ultimately opted for a simple thresholding method due to its superior performance. However, thresholding lacks reliability across all video inputs, making the gradient-based approach more suitable for real-world applications. The RMBG segmentation is still available in the final pipeline in order to provide a ”quality” mode where one can prioritize quality and robustness over performance.

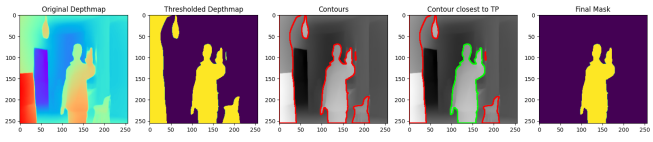

Fig. 3. Foreground Segmentation: Contour based approach.

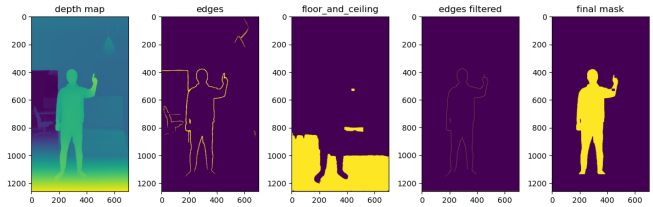

Fig. 4. Foreground Segmentation: Gradient based approach.

Point Cloud Estimation

Once the depth map and person mask are obtained, the point cloud is generated by directly mapping depth intensity values to 3D positions. Each pixel in the depth map is transformed into a corresponding 3D point, using its depth intensity as the Z-coordinate.

To ensure a clean and usable point cloud, various filtering techniques are applied. Noise, outliers, and unconnected clusters, often caused by segmentation errors, are removed. Specifically, the Open3D [9] functions *remove_statistical_outlier* and *remove_radius_outlier* are employed to eliminate small inconsistencies, while the DBSCAN clustering algorithm is used to discard larger background artifacts. These post-processing steps significantly enhance the quality of the final point cloud, ensuring a stable foundation for subsequent reconstruction and meshing.

Fig. 5. Results of the Point Cloud Estimation Modules.

V. Point Cloud Reconstruction & Our ML Model

In this section of the pipeline, we artificially reconstruct the parts of the human body, which were not captured by the camera, by using our self-trained custom neural network.

A. Common Architectures

Common deep-learning architectures for point clouds are point-based, graph-based, sparse voxel-based, spatial CNN-based, or transformer-based architectures [10]. Since our goal is real-time performance, we did not consider complex and deep structures such as transformers as a viable option. Data structures like Truncated Distance Fields (TDF) showed promise, but were computationally and memory intensive, while their results were comparable to a simpler voxel-based approach.

Consequently, we opted for a fast, intuitive, and dynamically adaptable method based on voxels paired with a 3D convolutional neural network.

B. Dataset

For training, we used the THuman2.1 dataset [11], which contains 2445 highly detailed human meshes. In addition, we generated 2776 custom poses by animating[3] a human dummy mesh[4] and baking new meshes from it, resulting in a total of 5221 meshes. Each mesh was converted into a point cloud with 10,000 points using the Open3D [9] library.

C. Data Preperation

Before being passed to the model, the point clouds are converted into a voxel grid. For that, they are centered within a vertical bounding box from (0,0,0) to (64,128,64) and normalized accordingly. The vertical shape of the bounding box was chosen, because our primary use case are persons standing upright in front of a camera. With the vertical bounding box setup we are achieving additional resolution in the vertical direction.

After normalization, each point cloud is voxelized to a voxelgrid with a grid size of 1 and then transformed into a 3D volume tensor, which is fed into the model. During training, random data augmentations such as mirroring, rotations and translations along the vertical axis are applied. This ensures robustness to various orientations and accounts for scenarios where, for example, the camera does not fully capture the subject’s legs. To simulate point clouds captured from a single view, we placed a virtual camera and cut all points which were not visible from that view, with the help of the Open3D [9] library. An example of a model input can be seen in Fig. 7. The cut version of the point cloud is used as input and the full version of the point cloud is used as the target for the training.

[3]mixamo.com
[4]assetstore.unity.com/3d

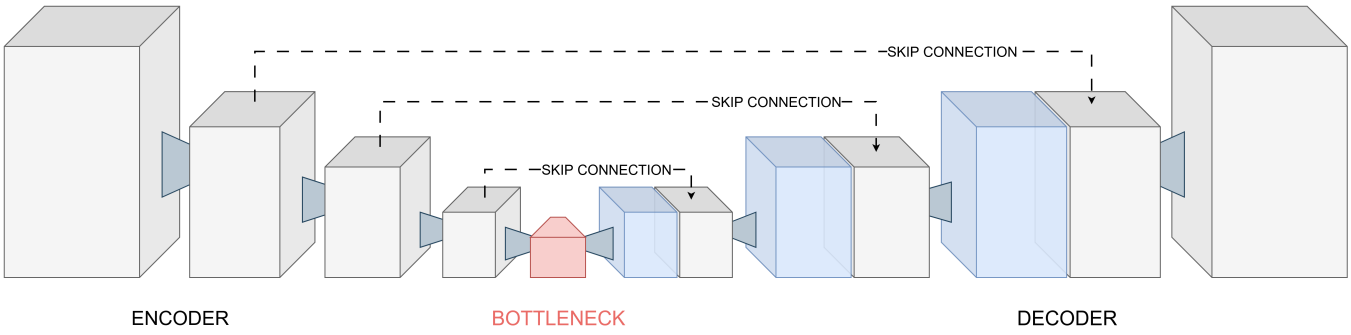

Fig. 6. U-Net style autoencoder for reconstruction of incomplete point clouds. Each step includes two consecutive calls of a 3D Convolution followed by Batch Normalization and ReLU activation. In the encoder part these steps are then followed by a max pooling for downsampling. In the decoder part these steps are followed by a transposed 3D convolution with stride 2 for upsampling. Sigmoid activation function is applied on the output to achieve values between 0 and 1.

Fig. 7. Normalized point cloud within a vertical 64x128x64 bounding box. Point cloud will be voxelized and then transformed to a tensor volume as an input for the reconstruction neural network.

D. Model Architecture

The model used for reconstruction is an autoencoder in a U-Net styled architecture Fig. 6. The encoder and bottleneck components allow the model to extract essential human structure and pose information. The decoder then reconstructs the full human figure, including the missing regions. Skip connections help retain information about the initial position and shape of the input, leading to significantly better results compared to a conventional autoencoder design.

The model output represents the certainty that any given voxel in the grid should be labeled as "occupied" (i.e., an expected body point). For training, we used a binary cross-entropy loss function with a specialized weighting strategy.

$$l_n = -w_n[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)] \quad (1)$$

Since the target point clouds only consists of 10,000 points, whereas the 64×128×64 voxel grid has 524,288 cells, we increase the loss weight in the area around the expected body points. The closer the cell is to an expected point, the higher the additional weight. The additional weight prevents the model from trivially setting most voxels near-zero. As a result, the model is driven to produce higher occupancy values, which simplifies thresholding and yields sharper reconstructions with less holes, which were observed in results of previously trained models without the additional weight.

E. Postprocessing

After thresholding, a non-maximum suppression step is performed to drastically reduce the number of occupied voxels prior to mesh creation, to decrease computational overhead. Instead of examining all neighboring cells in a 3D volume, our non-maximum suppression considers only neighbors along the three directional axes. An visual example of our non-maximum suppression can be seen in Fig. 8. This approach balances computational efficiency with the ability to preserve relevant structure. The remaining cells with a value greater than zero are considered as part of the reconstructed point cloud. An example of a reconstructed point cloud after post processing can be seen in Fig. 9

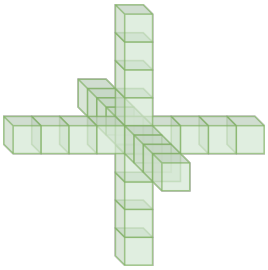

Fig. 8. Non-maximum suppression: The centered cell is compared with the neighbors along the three axes. If the cell is the maximum in either of these axes, it keeps its value, otherwise the cell will be set to zero.

F. Speed

Due to the small size of our model, it only takes a few milliseconds to complete a point cloud. Thanks to parallelization with CUDA [12] and PyTorch [13], the entire reconstruction process takes only about 50 to 100 milliseconds on our system. Depending on preference, it is possible to further reduce or increase the number of features in each layer.

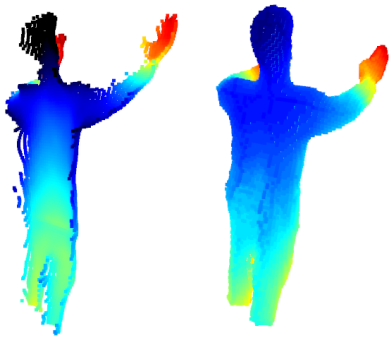

Fig. 9. Example of our point cloud reconstruction. On the left side is the incomplete point cloud with an open back and on the right side the result of our reconstruction after post processing.

VI. Mesh Generation

In the mesh generation step the mesh is created from the previously reconstructed point cloud. There are multiple approaches for surface reconstruction. Many of them require normals for the points of the point cloud.

A. Normal Estimation

Since our reconstructed point cloud does not include any normals, we use the Open3D [9] function *estimate_normals* to calculate them. Then each point gets a direction as seen in

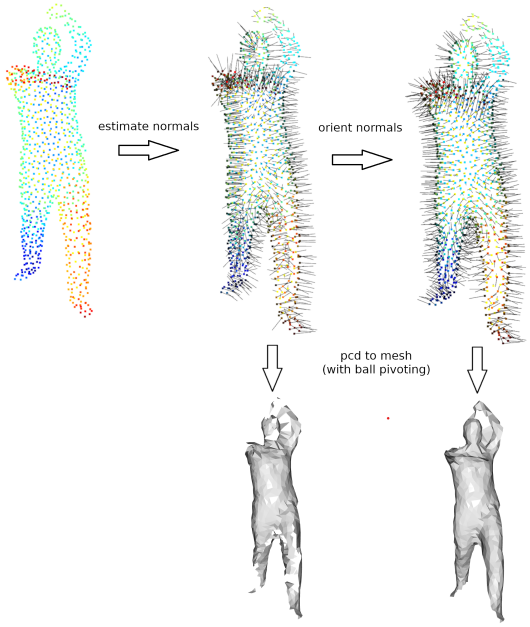

Fig. 10. Normal Estimation and Orientation

the center image in Fig. 10. To orient the normals properly we use the function *orient_normals_consistent_tangent_plane*, which uses a minimal spanning tree. However, orienting the for a human shape is not a simple task. The human body has complex parts like hands or feet where orienting normals becomes difficult. Either there are not enough normals to achieve an isotropic orientation, or it conflicts with neighboring normals, leading to suboptimal alignment. On the right of Fig. 10 is a point cloud after an additional normal orientation step. An additional orientation step clearly improves the quality of the resulting mesh, but it is still not perfect. To improve our meshes we apply orientation steps in approaches like Poisson, Ball Pivoting.

B. Mesh Generation Approaches

To generate meshes, we have tried different approaches. New Machine Learning approaches like point2mesh [14] were too slow or sometimes not openly available. Therefore, we use more established methods. Open3d [9] provides some methods for reconstructing surfaces[5]. Scikit-Image [15] implements marching cubes.

Currently, our software supports the mesh generation with four approaches:

- Alpha Shape
- Poisson
- Ball Pivoting
- Marching Cubes

In the following, the approaches are shortly described.

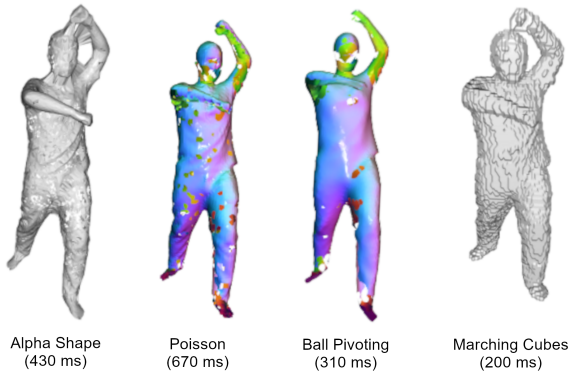

Fig. 11. Mesh Generation Approaches

Alpha Shape
The alpha shape approach uses a set of points to create a more precise form than the convex hull. [16] It uses the generalized disk radius α as a tradeoff parameter. With higher α the mesh is more connected. With lower α the mesh is more detailed, but has holes. In Fig. 11 we use $\alpha = 3$ which seemed like the best compromise.

Poisson
Poisson considers the surface reconstruction with oriented points as a spatial Poisson problem and solves this normalization problem. [17] A higher depth parameter results in

[5] https://www.open3d.org/html/tutorial/geometry/surface_reconstruction.html

more detailed results, but requires more time. In Fig. 11 we define $depth = 9$.

Ball Pivoting
Ball pivoting creates a triangular mesh by pivoting a ball through the points. With the radius of the ball the density is defined. [18] In Fig. 11 we define the radius dependent on the average distance between the points and use multiple products of that.

Marching Cubes
Marching cubes converts voxel data into a polygonal representation. It goes through the cubes (formed from eight voxels) and calculates the best shape for the voxel. [19] This approach fits well with structure we use in the point cloud reconstruction, since the data is already voxelized, as we described in V-C.

Open3d [9] and Scikit-Image [15] implement these approaches. Our software applies them with corresponding configurations. The user can choose their preferred method, otherwise marching cubes is set as default. The software can visualize the generated meshes and there is also the option to stream the mesh to different platforms.

VII. Streaming and Unity Visualization

This section presents the streaming of the results and the Unity visualization. The reconstructed point cloud and mesh can be streamed to Unity.

We decided to stream the results via web sockets. Web sockets are interoperable and easy to include in various platforms. The streaming process runs in a separate thread. Therefore, it does not interfere with the performance as much.

Streaming via Web Sockets
To stream the results the pipeline uses the python Socket[6]. The two modules *MeshStreamer* and *PcdStreamer* in the pipeline handle the streaming task.

MeshStreamer streams the mesh as a .obj standard data string. This contains vertices, normals and triangles. *PcdStreamer* streams the point cloud as a .ply standard data string. This contains all point coordinates of the point cloud. The data is sent as an encoded byte string.

Our Streaming Mechanism appends 4 control bit to the data string, which contain the length of the data. we use this in Unity to check if the received data is complete and valid, since the websocket packages we use do not guarantee a full data transfer per package.

Unity Visualization
In Unity we created a small sample scene. As mentioned before, both the point cloud and Mesh are streamed to unity will be rendered here. To receive data from Python, the script uses the Network Stream Class[7]. In Fig. 12 you see the sample scene with one point cloud and its mesh visualized.

Fig. 12. Our Unity sample scene that visualizes the results of the python pipeline

VIII. Outlook

Since this project was limited in time, there are still some points that could be dealt with in future extensions: **(1)** There are no temporal, or frame-to-frame, connections built in in the pipeline. Each frame is treated as a single new frame without being aware of it's past. This causes some shaking and flickering in the final result. **(2)** The foreground segmentation solution we have chosen is fast, but not reliable on different scenarios than our provided demo recording. We did provide a stable solution with the RMBG Model [8] but this dramatically slows down the pipeline. A new implementation that has a good trade off between quality and reliability should be considered.

IX. Results and Conclusion

To conclude, we presented a fully working start-to-end pipeline solution that converts simple 2D Video Streams into a waterproof 3D Mesh that can be used in any 3D Software. One of our initial goals was a close to real-time solution, and we can proudly say we did keep the processing times of each module to a very small scale and the pipeline produces 2-3 frames even on moderate personal laptops. Even though our pipeline does not consider any temporal information in between frames, the results seem rather smooth and are certainly presentable. With under 100 ms per frame our custom trained U-Net run with acceptable speed while still providing very satisfying point clouds as results. A special configuration using a Flying Edges[8] algorithm for mesh reconstruction and some shortcuts achieved a consistent runtime of 200-300 ms for the whole pipeline on our fastest desktop PC. Comparing to other similar works, as presented in II, we have to admit to not have the best output. Here, we realized just how effective and high-performing an end-to-end deep learning approach can be. A major drawback of our

[6] https://docs.python.org/3/library/socket.html
[7] https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets.networkstream?view=net-9.0
[8] vtk.org

pipeline is, that each step needs its own computational time, and any errors or inaccuracies propagate to subsequent steps. But we also have to mention that our pipeline, compared to others, is able to operate on one single 2D image without any pre-prepared depth or scene information. The only assumption we have is, that a human exists in the recorded scene. The final results and a demo video can be found in the provided git repository and a screenshot can be seen in the Appendix (see 13).

REFERENCES

[1] A. Kanazawa *et al.*, "End-to-end recovery of human shape and pose," in *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[2] T. Alldieck *et al.*, "Tex2shape: Detailed full human body geometry from a single image," in *IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2019.

[3] Z. Zheng *et al.*, "Deephuman: 3d human reconstruction from a single image," in *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[4] S. Saito *et al.*, "Pifu: Pixel-aligned implicit function for high-resolution clothed human digitization," in *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[5] S. Saito *et al.*, "Pifuhd: Multi-level pixel-aligned implicit function for high-resolution 3d human digitization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[6] L. Yang *et al.*, "Depth anything v2," *arXiv:2406.09414*, 2024.

[7] Meta, *Sam: Segment anything model*, Accessed: 2025-02-26, 2025. [Online]. Available: https://segment-anything.com/.

[8] BriaAI, *Rmbg-1.4: Robust background removal model*, Accessed: 2025-02-26, 2025. [Online]. Available: https://huggingface.co/briaai/RMBG-1.4.

[9] Q.-Y. Zhou *et al.*, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[10] A. Xiao *et al.*, "Unsupervised representation learning for point clouds: A survey," *arXiv preprint arXiv:2202.13589*, 2022.

[11] T. Yu *et al.*, "Function4d: Real-time human volumetric capture from very sparse consumer rgbd sensors," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR2021)*, 2021.

[12] T. Besard *et al.*, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018, ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[13] A. Paszke *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1912.01703.

[14] R. Hanocka *et al.*, "Point2mesh: A self-prior for deformable meshes," *ACM Trans. Graph.*, vol. 39, no. 4, 2020, ISSN: 0730-0301. DOI: 10.1145/3386569.3392415. [Online]. Available: https://doi.org/10.1145/3386569.3392415.

[15] S. J. van der Walt *et al.*, "scikit-image: image processing in Python," *PeerJ*, vol. 2, e453, Jun. 2014. DOI: 10.7717/peerj.453. [Online]. Available: https://doi.org/10.7717/peerj.453.

[16] H. Edelsbrunner *et al.*, "On the shape of a set of points in the plane," in *IEEE Transactions on Information Theory*, 1983. [Online]. Available: https://api.semanticscholar.org/CorpusID:6983029.

[17] M. Kazhdan *et al.*, "Poisson Surface Reconstruction," in *Symposium on Geometry Processing*, A. Sheffer *et al.*, Eds., The Eurographics Association, 2006, ISBN: 3-905673-24-X. DOI: /10.2312/SGP/SGP06/061-070.

[18] F. Bernardini *et al.*, "The ball-pivoting algorithm for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 4, pp. 349–359, 1999. DOI: 10.1109/2945.817351.

[19] W. E. Lorensen *et al.*, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87, New York, NY, USA: Association for Computing Machinery, 1987, 163–169, ISBN: 0897912276. DOI: 10.1145/37401.37422. [Online]. Available: https://doi.org/10.1145/37401.37422.

APPENDIX I
ADDITIONAL GRAPHICS

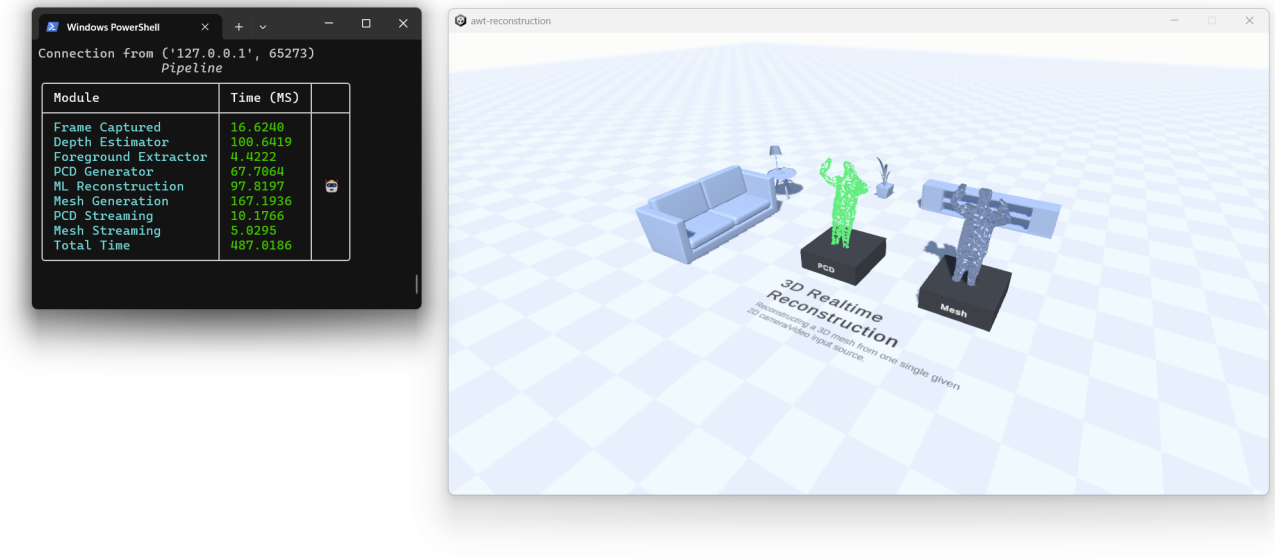

Fig. 13. The Results of the Pipeline visualized in Unity (right) and the performance measurements of each module seen in our console interface (left)