Face Reconstruction

Alexander Mueller[1], Arda Keskiner[1], John Flynn[1], and Vitalii Rusinov[1]

[1]Technical University of Munich

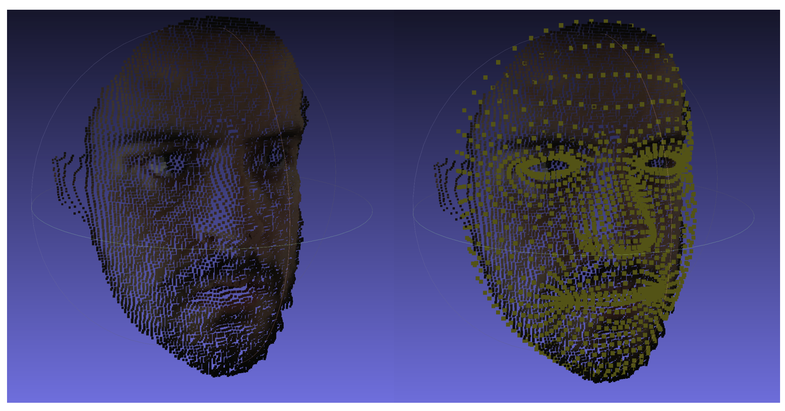
Introduction

In this project we performed face reconstruction following the work of Thies et al. [1]. We were fitting a parametric face model to incoming RGB-D scans taken with a Kinect sensor. Facial landmarks allowed us to align the scan with the parametric model initializing the transformation matrices. Afterwards, we optimized the energy formulation w.r.t. to the fitting and regularization terms. The final result is submitted to a real-time rendering solution that allows to manipulate the final result parameters.

Methods

Mesh Preprosessing

At first we input the noisy point cloud of the face using Microsoft Kinect API. The point cloud is then filtered using subsampling and statistical outlier removal. The mesh is reconstructed using a chosen surface reconstructed algorithm (Poisson reconstruction, Ball Pivoting, Marching cubes) and is then further refined. Keypoints of the face are tracked using the Microsoft HD Face Tracker. Keypoints on the mouth corners, nose and forehead are manually associated with vertices in the face model and serve as initial sparse correspondence points.



Optimization

The following model produced by Thies et al. [1] is used to characterize our face model

$$\mathcal{M}_{\text{geo}}(\boldsymbol{\alpha}, \boldsymbol{\delta}) = \boldsymbol{a}_{\text{id}} + E_{\text{id}}\boldsymbol{\alpha} + E_{\text{exp}}\boldsymbol{\delta},$$

Where $\boldsymbol{\alpha}$ and $\boldsymbol{\delta}$ are the PCA components of face geometry and expression, respectively and $\mathcal{M}_{\text{geo}}(\boldsymbol{\alpha}, \boldsymbol{\delta})$ is a stacked vector of vertices of the model. Additionally, the rigid transformation $[\boldsymbol{R}, \boldsymbol{t}]$ of the Kinect scan is modeled. We solve for $\{\boldsymbol{\alpha}, \boldsymbol{\delta}, \boldsymbol{R}, \boldsymbol{t}\}$ simultaneously by minimizing the sum-of-squares error $E_{point} = \sum_i (\boldsymbol{v}_{j(i),scan} - \boldsymbol{v}_{i,model})^2$ between correspondences, where i, j are face and scan vertex indices, respectively.
Parameters $\{\boldsymbol{\alpha}, \boldsymbol{\delta}, \boldsymbol{R}, \boldsymbol{t}\}$ are found in the following iterative process:

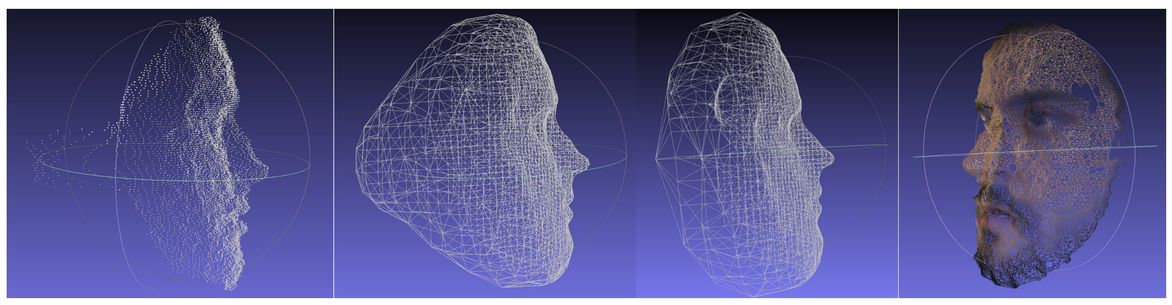
1. Minimize the sum-squared-distance error for sparce correspondence points (Kinect HD Face).
2. Find the nearest scan vertex (1-KNN) to each model vertex.
3. Minimize the sum-squared-distance error E_{point} for correspondence points found via KNN below a distance threshold.
4. Repeat step 2 for a set number of iterations or until convergence.

In the energy terms we include the geometric fitting terms for sparse, dense and keypoint correspondences that also depend on the vertex normals as well as the regularization terms.

Rendering

The custom rendering solution is based on the Vulkan API v.1.1.108. Using a custom API allows for an easy, high-performance integration of the optimization code into the rendering back-end. As is common in render applications, the render code itself runs in an additional thread to prevent slowdown by the other calculations. As a result, we can achieve a smooth frame-rate while giving the optimization the time necessary to create a good result.
This low-level access to the rendering back-end enables us to adjust the settings and features of the API to perfectly match our desired results. Some examples for those features are Anti-Aliasing, Culling and setting up custom constant buffers (i.e. buffers containing the mesh data for the shaders in the rendering stage). Since our file formats rely on colour information being stored in the vertex descriptions, customized buffers and shaders allow us to easily adjust all steps needed for the result.
One more major advantage of Vulkan is its support for a variety of operating systems. Since some team members were using Mac OS and some Windows, using Vulkan in combination with an additional layer called MoltenVK enabled us to work on the project without specific device requirements. Additional libraries might require fallback layers in the future, however having the render back-end adjustable makes expanding the software in the future easier. On top of Vulkan, a GUI layer called "Dear IMGUI" was added to provide support for real-time interactions with the application. This allows for the easy creation of GUI elements and makes it easy to add additional functionality to the application in the future.

Results



To maximize the mesh quality of appearance we chose a threshold of 4mm between valid correspondence points and a weight of 100x for sparse correspondences. Border vertices in the scan mesh were excluded because their depth values are often inaccurate.

To reduce computation time we optimize only over vertices on the front face of the model (e.g. neck and ears are excluded) and of these vertices we randomly sample 50% of them to create residuals. We experimentally found that optimizing over the first 40 geometric and expression PCA components preserved the quality of the face model.

Above left is a Kinect scan of a face with the 1st, 4th and 20th iteration of the optimizer. This solution was generated in 15.86 seconds on a 2015 Macbook Pro, though we experimentally found that the face model converges after roughly 6 iterations in 6.84 seconds.

Conclusions

In this project we reconstructed face surfaces and performed geometric face model fitting. We extracted face meshes from a Kinect scan and selected sparse correspondences at easy-to-identify positions on a face. We then used a human face model to fit the pose, geometry and expression of the face scan with accurate and efficient results on a commodity CPU. Finally, we developed a renderer to visualize our model. While we are satisfied with our results, our reconstruction can be improved by modeling albedo and our optimizer could be redesigned to run on a GPU.

Acknowledgements

We used OpenMesh and Eigen libraries to import and perform linear transformations on model vertices, PCL to filter the point clouds, Ceres to minimize the sum-of-squared-distance error, Sophus to model the rigid transformation, Nabo to find nearest neighbors and Vulkan for our rendering. We are grateful to Dr. Justus Thies for continuous supervision and support.

References

[1] J. Thies, M. Zollhöfer, M. Nießner, L. Valgaerts, M. Stamminger, and C. Theobalt. Real-time expression transfer for facial reenactment. *ACM Transactions on Graphics (TOG)*, 34(6), 2015.