TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

Real-Time Image-Based 3D-VR-Rendering of Open Environments

Masterarbeit im Studiengang Informatik

vorgelegt von

Bastian Kuth

Matrikelnummer 3609842

Erstgutachter:	Prof. Dr. rer. nat. Bartosz von Rymon Lipiński
Zweitgutachter:	Prof. Dr.-Ing. Christian Schiedermeier

Abstract

The most common objective in rendering is to achieve photo-realism: Synthetic images should look like they were taken from a real camera. To accomplish this, today's photo-realistic renderers either simulate the complex physical phenomenon of light traveling through a scene or approximate novel views, e.g. by training an image-generating neural network with a large number of photographs taken from similar perspectives. Both approaches suffer from high computational costs, with recent graphics hardware advances trying to speed up the computation time to achieve real-time rendering. Furthermore, geometry-based renderers, e.g. Monte-Carlo pathtracers, require artist-authored 3D models, materials, and textures.

Image-based rendering is a method for synthesizing novel views directly from photographs. Therefore, when disregarding rendering artifacts, the resulting synthesized images are already photo-realistic by definition. In this thesis, we present an image-based six degree of freedom renderer which synthesizes images from a small number of photographs in real-time. Given datasets of densely feature-matched photographs with reconstructed camera poses, we build an *image graph* data structure: To enable six-degree-of-freedom movement by the user, relevant images that span a simplex around the desired virtual camera position are combined. By having multiple of these simplexes with photographs taken from different angles, we allow for camera rotation. The resulting renderer runs fast enough for consumer grade hardware and is *virtual reality* (VR) compatible.

Additionally, we propse a GPU-based genetic and brute-force optimization technique to achieve visually improved image morphing results for feature matching. Instead of a pure GPGPU based approach, we make direct use of the rendering pipeline and rasterization hardware of graphics cards to optimize a visual error metric. We show that our technique improves morphing meshes to a sub-pixel precision for examples from image-based rendering.

Contents

List of Figures

List of Tables

List of Acronyms

CNN *convolutional neural network*

ECC *enhanced correlation coefficient*

GPGPU *general purpose computation on graphics processing unit*

GPU *graphics processing unit*

IBMPP *image-based media production pipeline*

IBR *image based rendering*

SSIM *structural similarity index measure*

VR *virtual reality*

1 Introduction

In traditional computer graphics, virtual worlds are typically described by polygon-based 3D models. Despite being the industry standard for rendering, and thus subject to large amounts of research, funding, and development, real-time 3D mesh-based approaches still lack to fulfill the ultimate goal of efficient photo-realistic image synthesis. Additionally, the creation of high quality 3D assets is a time-consuming, manual, and potentially expensive process, as it requires trained experts.
One possible alternative solution to the 3D mesh-based approach is *image based rendering* (IBR). IBR requires technologies from the fields of computer vision, image processing, and computer graphics. Given a set of images of a scene, IBR techniques can generate novel views within this scene by directly or at least implicitly sampling from the scene images.
The *image-based media production pipeline* (IBMPP) is a project at the *Game Tech Lab* of TH Nürnberg to advance the concepts of IBR. This project aims to create an image-based and as far as possible geometry-free processing chain for photorealistic 3D applications for games, virtual art and culture exploration, digital twins for architecture and industry, and for photogrammetry.

1.1 Challenges

Goal of this thesis is to develop a real-time, *graphics processing unit* (GPU) accelerated rendering application for displaying given IBMPP datasets and prototype novel techniques to support and to further the scientific work of the project. The application should be fast to run, easily expandable, and enable six degrees of freedom of movement and rotation. Furthermore, a use of the renderer in virtual reality is required. GPU acceleration should be handled using a modern version of the OpenGL API. Additionally, this work explores a method of how the dataset creation process can profit from GPU accelerated processing.
As the computer vision side of the rendering technique, thus preprocessing and dataset creation, is subject of current and future research by different members of the IBMPP project, this thesis does not give any detailed explanation or analysis about it.

1.2 Structure

In agreement with the advisor Prof. Dr. von Rymon Lipiński the structure of this thesis is more research-oriented than the norm. To be more in line with today's scientific publications, this

thesis is a collection of articles, similar to a compilational doctoral thesis, instead of a monograph. Therefore, section 2 and section 3 are written as fully independent articles and follow the structure of a paper. Thus, they have their own abstract, introduction, bibliography etc., and do not require knowledge from other sections. Instead, these sections require the prior knowledge of a professional audience from the field of visual computing. Section 4 covers additional noteworthy emerged results of this thesis, like software architecture and tools, followed by a conclusion in section 5.

2 View Interpolation Revisited in VR

2.1 Abstract

We revisit and expand traditional methods for image-based rendering to enable the exploration of static and open environments in Virtual Reality. Given a sparse set of real or synthetic views of a scene as input, our pipeline builds a rendering data structure called *image graph*, which consists of morphing cells. Each cell consists of multiple real views, which can be morphed together to synthesize novel views without requiring depth information. Our rendering algorithm allows for six degrees of freedom of movement and can produce images with a different field of view than the source images. Neighboring cells are extrapolated to fill in missing or incorrect information. Additionally, we use real-time histogram specification to counteract exposure inconsistencies within a dataset. Our method achieves rendering times of under 4ms using current hardware.

2.2 Introduction

With the introduction of VR headsets into the consumer market, combined with major advances in the field of machine learning, there is a resurgence of research in *image-based rendering* (IBR). Given a set of images of a scene, IBR techniques can synthesize novel views within that scene. While the 3D geometry of a set of photographs can be approximated using structure from motion [1–3], new views rendered using this reconstructed geometry usually do not appear photorealistic. This is because rendering the densely reconstructed point clouds directly with splatting [4] can suffer from hole filling issues and blurry results. Alternatively, reconstructed point clouds can be triangulated into meshes, but techniques often struggle to reconstruct featureless, reflective, or refractive surfaces. Instead, IBR methods focus on rendering by directly, or at least implicitly, sampling from the given set of images. Neural-network-based IBR methods utilize the input images to create an abstract scene representation to later render novel views from it.

In this work, we revisit traditional methods for IBR, transform them for modern hardware using the OpenGL graphics API, and extend them for a six degree of freedom of movement, and propose a novel parallax estimation technique. Hereby lies our focus on scenes of open environments with only distant parallaxes. Additionally, we propose a method for handling color inconsistencies within a dataset. In detail, the following contributions are made:

- We describe an automatic, image-based and depth-less *preprocessing pipeline* for creating

datasets of large, static, and open environments.

- We present a highly efficient *rendering algorithm* that can be used for virtual reality applications.
- We define a unique *rendering order* to enable all six degrees of freedom of movement, virtual focal length, and the handling of simple parallaxes.
- We show how to mitigate the *problem of color inconsistencies* within a dataset in real-time.

2.3 Related Work

For over 30 years, researchers aimed to find techniques to synthesize novel views from a given set of input images. This work revisits the traditional approaches by Chen [5], and Chen and Williams [6]. In the latter publication, the authors describe the interpolation between views using *image morphing* for an alternative approach to 3D rendering. Image morphing was established by Beier and Neely [7] as the process of transforming or warping the elements of one image into the other image and simultaneously blending the color values to approximate in-between views.
Later, the concepts were extended to be used with virtual reality in [5]. By arranging panoramic views into a grid and snapping or interpolating between the closest grid points to the virtual camera, the technique allows for virtual translation. Rotation is achieved by rotating the panoramic views accordingly. The described concepts are a precursor to our image graph data structure.

2.3.1 Other Traditional Methods

McMillan and Bishop [8] pioneered in the field by connecting image-based rendering with the plenoptic function. A model which was later utilized by Gortler et al. [9] and Levoy et al. [10] to construct data structures inspired by the plenoptic function to represent a scene. Seitz and Dyer [11] adapt morphing specifically for rendering camera and object motion. By approximating minimalistic meshes combined with projective texture-mapping and hole filling, Debevec et al. [12] present a hybrid between IBR rendering and structure from motion reconstruction. Chaurasia et al. [13] warp superpixels based on a depth approximation. Bertl et al. [14, 15] present a technique for reconstructing parallax-capable panoramas that can be viewed in VR. Several approaches such as [16,17] exist that make use of depth information, e.g. recorded with special RGBD cameras or from reconstruction to render novel views.

2.3.2 Neural Based Methods

Hedman et al. [18] present a deep learning approach for blending in the context of IBR. They train a *convolutional neural network* (CNN) to learn blending weights for combining input image contributions for synthesizing a new image. Recent works [19–21] train a network to dissect the input images into multiple planes (MPIs) of different elements and depth. This enables them to more correctly transform the image elements than previous techniques. Mildenhall et al. [22] demonstrate that a deep neural network can be trained to represent the radiance of an object from a given view. This technique is called *neural radiance fields*. Extensions of this concept have been published, e.g., by Liu et al. [23], who utilize a voxel data structure to speed up rendering. Rückert et al. [24] use a hybrid approach by reconstructing and rendering a point cloud from a given dataset, but augmenting it with learned feature vectors. For rendering, they rasterize the point cloud and use a deep neural network to fill the remaining gaps between the one-pixel splats. Additionally, their approach contains a physically-based photometric camera model to handle input images with varying exposure and white balance.

2.4 Image Graph

In this section, we describe the creation process of our image graph data structure from a set of images. Additionally, an image acquisition technique to achieve the best possible results is proposed.

2.4.1 Graph Creation

In our renderer, a scene is represented by the *image graph* data structure. To create such a graph, we first reconstruct the camera poses of the set of input images, using standard structure from motion techniques. Together with an estimated camera pose, each input image is a node of the graph. Nodes with a similar pitch and yaw rotation that span a simplex are triangulated into a *morphing cell*. Depending on the desired degrees of translation freedom, the simplex is a line, a triangle, or a tetrahedron.

Within each morphing cell, a dense feature matching technique to find matches between the *members* of the cell is performed. In our implementation, we use the matching technique based on homographic decomposition by Seibt et al. [25]. Subsequently, once per cell, the matched feature points are triangulated with Delaunay triangulation [26] into a mesh. We perform the triangulation on the average pixel position of each feature point. Thus, all members of the cell receive the same triangulation. By interpolating the positions of the mesh barycentrically

and simultaneously blending the colors with the barycentric coordinates, we can synthesize a novel view inside the cell. Triangles of the morphing mesh that do not contain the same image information in all the used images, e.g. because they are affected by a parallax or an incorrect feature match are filtered out. We base the filtering of each triangle on the homogeneous-inhomogeneous criteria of the used feature matching technique [25]. Alternatively, or in addition, other image alignment metrics such as the *structural similarity index measure* (SSIM) [27] or the *enhanced correlation coefficient* (ECC) [28] can be used on a triangle basis as a filtering criterion.
In case the input images follow a panoramic projection, such as a 360° equirectangular projection, we perform the feature point related computations on multiple reprojected pinhole images and project the results back onto the sphere. To triangulate the key points on a sphere, the spherical Delaunay triangulation by Caroli et. al [29] is used.

2.4.2 Image Acquisition

We support both *ordered* and *unordered* image acquisition, while the latter means a set of images taken without any special strategy. For best results, we recommend an ordered acquisition: First, the user takes multiple, slightly overlapping images on a single spot, while rotating the yaw and pitch of the camera. The number of images per spot depends on the focal length (field of view) of the used camera and the desired field of view inside the renderer. This step is repeated for all the corners of a simplex grid that fills the space of the desired freedom of movement. When operating at foot-height, we recommend a grid spacing between 0.5 and 2 meters. For aerial shots, or for shots with only distant scenery, we found that much larger distances can be sufficient. Our technique can also be applied to panoramic images. In case the panorama fills the whole sphere, only a single image per grid corner is required.
After processing such an ordered image graph, multiple cells lie directly on top of each other, each representing a rotation. This is beneficial, because when moving between cells of the graph in the renderer, the cell switch might be visible, depending on the accuracy of the image processing. Stacked rotational cells result in a single switch of all the cells at once, which results in an overall smoother rendering.
Additionally, we support *synthetic* image acquisition, thus images generated with a different, e.g. non real-time rendering technique. Here, we recommend to generate a single 360° panoramic image per grid corner. In case the synthetic method does only allow for pinhole cameras, we recommend to generate six rotational images arranged in a cubemap fashion (up,

down, forward, backward, left, right). As preprocessing might generate a morphing mesh that does not fill the whole image size, we suggest to use a larger field of view than the theoretically required 90° of a cubemap. We found a field of view between 100° and 110° to be sufficient.

2.5 Rendering

In this section, we describe how to render a single cell of our image graph from the perspective of a virtual camera. Next, we motivate and describe the rendering process of multiple cells. Furthermore, we describe our color mapping technique and the required modifications for our technique to be used in VR.

2.5.1 Single Cell Morphing

To render a single cell of the graph, the following process is used: First, we find the barycentric coordinates of the desired 3D *virtual camera* position for each cell. This is trivial when working with tetrahedrons as morphing cells. Otherwise, when working with simplices of a lower dimension than the three degrees of translational freedom of the virtual camera, we search for the barycentric coordinate that represents a position in the domain of the simplex that is closest to the virtual camera position. The following equations are required: In case the simplex is a line between $\vec{a} \in \mathbb{R}^3$, and $\vec{b} \in \mathbb{R}^3$, the closest normalized barycentric coordinate $\beta \in \mathbb{R}$ for $\vec{b}$ of a point $\vec{p} \in \mathbb{R}^3$ is given by

$$\beta = \frac{(\vec{p} - \vec{a}) \cdot (\vec{b} - \vec{a})}{(\vec{b} - \vec{a}) \cdot (\vec{b} - \vec{a})}, \tag{1}$$

where $\cdot$ is the dot product. The closest normalized barycentric coordinates $\begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^T$ of a point $\vec{p} \in \mathbb{R}^3$ for a triangle defined by its corners $\vec{a} \in \mathbb{R}^3$, $\vec{b} \in \mathbb{R}^3$, and $\vec{c} \in \mathbb{R}^3$ are given by solving the system of equations:

$$\begin{bmatrix} \vec{a} & \vec{b} & \vec{c} & (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a}) \\ 1 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \nu \end{bmatrix} = \begin{bmatrix} \vec{p} \\ 1 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \nu \end{bmatrix} = \begin{bmatrix} \vec{a} & \vec{b} & \vec{c} & (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a}) \\ 1 & 1 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \vec{p} \\ 1 \end{bmatrix}, \tag{3}$$

where $\times$ is the cross product, and we ignore ν as the extra barycentric weight for the normal of the triangle plane.
Once the barycentric coordinates have been computed, the rendering process can begin.

Vertex Stage All the feature points of the morphing mesh of our cell are first handled in the vertex stage. The input attributes for each vertex are the two dimensional positions of the feature point in each of the cell member images. Augmented with the focal length of the image as a depth component and then normalized, the vertex shader combines all the pseudo positions into a single vector using the barycentric weights. This pseudo position represents the direction from which the light that created the feature point pixel came. We assume that the input images follow a pinhole model, and therefore do not have any lens distortions. Next, the combined pseudo position is transformed into world space. Lastly, we transform the position from world space into the space of the virtual camera, which can then be used to find the vertex position in the clip space of the used graphics API. We show this vertex stage in the equation:

$$E \cdot V \cdot \sum_{i=0}^{d-1} \left(\lambda_i \cdot C_i \cdot \frac{\vec{p}}{\| \vec{p} \|_2} \right), \tag{4}$$

where

- $\vec{p} = \begin{bmatrix} x_i & y_i & -f_i \end{bmatrix}^T$ is constructed from the feature point position in member image i and the image focal length. After the normalization, the vector acts as the pseudo position of the feature point.
- $x_i \in \mathbb{R}$ and $y_i \in \mathbb{R}$ are the positions of the feature point in member image i. The position is in camera coordinates, where $-\frac{w}{2} \leq x_i \leq \frac{w}{2}$ and $-\frac{h}{2} \leq y_i \leq \frac{h}{2}$. w and h are the width and height of the dataset images in pixels.
- $f \in \mathbb{R}$ is the focal length of member image i in pixels. After the normalization, it acts as a pseudo depth of a feature point. Since the camera is looking in the negative z-direction in our convention, we require the negative sign.
- C_i is the rotational transformation from a vector in camera coordinates of member image i to world coordinates, deduced from the reconstructed camera pose.
- $\lambda_i \in \mathbb{R}$ is the barycentric coordinate of member i.
- $d \in \{2, 3, 4\}$ is the dimension of the morphing cell.

- V is the virtual camera matrix, rotating a vector from world coordinates to the virtual camera coordinates.
- E is a matrix, transforming from camera coordinates into the clip space of the used graphics API. In case the pseudo depths fit into the bounds set by the near and far plane of the clip space, we can use the same projection matrix as for conventional 3D rendering.

For input images that follow camera models other than pinhole, the computation of the pseudo position has to be adjusted accordingly. For example, if the input images follow an equirectangular projection, $\vec{p}$ is given by:

$$\vec{p} = \begin{bmatrix} \cos(\chi_i)\sin(\psi_i) & \sin(\chi_i)\sin(\psi_i) & -\cos(\psi_i) \end{bmatrix}^T, \tag{5}$$

where $0 \leq \chi_i < 2\pi$ is the feature point longitude angle in member image i, thus the image coordinate iterating the width of the equirectangular image, and $0 \leq \psi_i < \pi$ is the latitude, thus the image coordinate iterating the height of the image.
Finally, we forward each of the 2D positions of the feature points as texture coordinates to the rasterizer. For 360° panoramas, a triangle $\triangle abc$ can have vertices on opposite sides of the image, e.g. when vertex a has a longitude of $\chi_a = 0 + \epsilon$ and vertex b has one of $\chi_b = 2\pi - \epsilon$. In such a case, the rasterizer would interpolate the spherical coordinates for a fragment lying in $\triangle abc$ the wrong way around the sphere. Therefore, when working with equirectangular panoramas, we pass the texture coordinates to the fragment stage in the 3D Cartesian representation, and renormalize and compute the actual texture coordinate in the fragment stage. This can be seen in fig. 1.

(a) equirectangular projection (b) spherical coordinates (c) cartesian coordinates

Figure 1: Panorama Texture Coordinate Interpolation. (a) Is our input 360° panorama image, which wraps around where the TV is located in the scene. (b) Passing over spherical coordinates results in incorrect sampling for triangles at the edge of the input image. (c) Passing over 3D coordinates instead, fixes the issue.

Fragment Stage In the fragment or pixel stage, we combine the color of each cell member, sampled from the images using their specific texture coordinate, either by blending, dithering or a mixture of the two. As we established a pseudo depth with the focal length earlier, the rasterizer takes care of a perspective-correct attribute interpolation, thus texture mapping.
When blending, extrapolation - as we do with the feature point positions - can result in values outside the valid color range. Therefore, for this rendering stage, the barycentric coordinates are clamped to the closest position inside the simplex. Note that this is not the same as clamping each individual barycentric weight between 0 and 1. Instead, values have to be clipped against the planes, edges, and corners of the simplex individually.
When the virtual camera is right in the center of a cell, imprecisions of the morphing mesh show the most, which results in a blurry image. To mitigate this, we further apply a smoothstep function [30] to the clamped barycentric coordinates. This gives the coordinates a bias towards a blend with a lower number of images. While smoothstep functions of any higher order can be constructed, we found that the function $f(x) = 6x^5 - 15x^4 + 10x^3$ suggested by Perlin [31] yields a good trade-off between image sharpness and a smooth transition while translating inside a cell. Applying a smoothstep function to normalized barycentric coordinates works by applying the function to each individual weight, and re-normalizing afterwards. In the following, $\hat{\lambda}_i$ represents the clamped and smoothstepped barycentric coordinate of source image i.
The blend color c_{blend} is selected from the source colors c_{i} of the cell members by the term

$$c_{\text{blend}} = \sum_{i=0}^{d-1} \left(\hat{\lambda}_i \cdot c_i \right). \tag{6}$$

When dithering, we apply the same clamping and smoothstepping as for blending. The dither color c_{dither} is selected from the source colors c_{i} of the cell members by the term

$$c_{\text{dither}} = \begin{cases} c_0, & \text{if } 0 \leq r < \hat{\lambda}_0 \\ c_1, & \text{if } \hat{\lambda}_0 \leq r < (\hat{\lambda}_0 + \hat{\lambda}_1) \\ c_2, & \text{if } (\hat{\lambda}_0 + \hat{\lambda}_1) \leq r < (\hat{\lambda}_0 + \hat{\lambda}_1 + \hat{\lambda}_2) \\ c_3, & \text{if } (\hat{\lambda}_0 + \hat{\lambda}_1 + \hat{\lambda}_2) \leq r < 1 \end{cases}, \tag{7}$$

where $0 \leq r < 1$ is a pseudo-random number. To reduce artifacts, we sample a tileable blue noise texture at the current screen coordinates for the pseudo-random numbers.

While blending can result in a blurry image, dithering can result in a grainy image. We found that mixing blending and dithering by taking the average of both can improve the perceived sharpness when working with real photographs of, e. g. trees or grass. An example of this can be seen in figure 2. On a VR device, the grainy appearance of dithering is much more visible, therefore here we recommend to use blending only.

(a) blend and dither

(b) blend (c) blend and dither (d) dither

Figure 2: Color Combination Methods. (a) Three input images are morphed into a single image using the combination of dithering and blending. The area encircled by the red square is magnified and shown with (b) dithering, (c) mixed dithering and blending, and (d) blending only. The images shows that blending is blurry, dithering grainy, and mixing both techniques a possible compromise.

2.5.2 Rendering Order for Multiple Cells

As blank regions in the viewport are undesirable - especially in virtual reality - our renderer tries to fill the viewport as much as possible. This is done by rendering all available morphing cells like the following: Given a virtual camera pose in three dimensions, we first compute its normalized barycentric coordinates for each morphing cell of the image graph. Next, we order the cells based on:

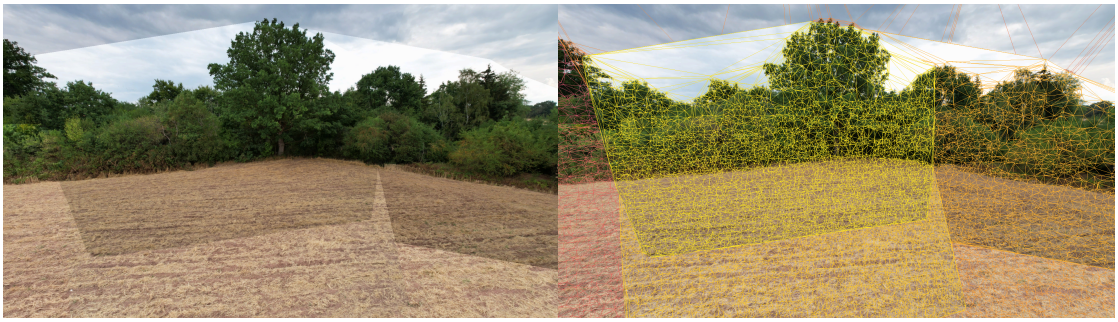
1. The *extrapolation distance*, i.e. the distance the cell needs to be extrapolated to represent the virtual camera translation. The distance is zero if the virtual camera pose lies within the cell.

2. The *rotational distance*, i.e. the angle between the virtual camera rotation (yaw and pitch) and the interpolated real camera pose.

The rotational distance comes into action when the extrapolation distance of multiple cells is the same, e.g. because multiple cells don't need to be extrapolated at all. This always happens in datasets with stacked rotational cells, acquired using the synthetic and ordered technique from section 2.4.2. Thus, the front-most cell, hereinafter referred to as the *main cell*, is drawn last and therefore overdraws the other cells. It requires the least extrapolation and has a rotation (yaw and pitch) that is most similar to the virtual rotation.
As an optimization, we ignore, thus cull, all cells that are outside the view frustum of the virtual camera. Additionally, it is useful to set a maximum allowed extrapolation distance. We don't draw cells that exceed this threshold. Figure 3 shows that extrapolating neighboring cells can fill some of the filtered triangles to approximate parallaxes. While extrapolation to positions close to a morphing cell works well, heavily extrapolated cells do not perfectly align with the main cell, thus reduce image quality. Figure 4 shows multiple cells stitched together to fill the viewport. In this example the transitions between the cells are visible due to exposure inconsistencies within the dataset. We address this problem in the following section.

(a) single cell (b) multiple cells

Figure 3: Parallax Estimation. (a) Rendering of a single cell can leave gaps, as we filter out triangles that can not be morphed properly, e.g. because of parallaxes. (b) Extrapolating multiple neighboring cells and adding them behind the original cell can fill some of the gaps. As a result, some of the extrapolated image elements do not perfectly align with the image elements of the main cell.



(a) morphed cells (b) wireframe overlay

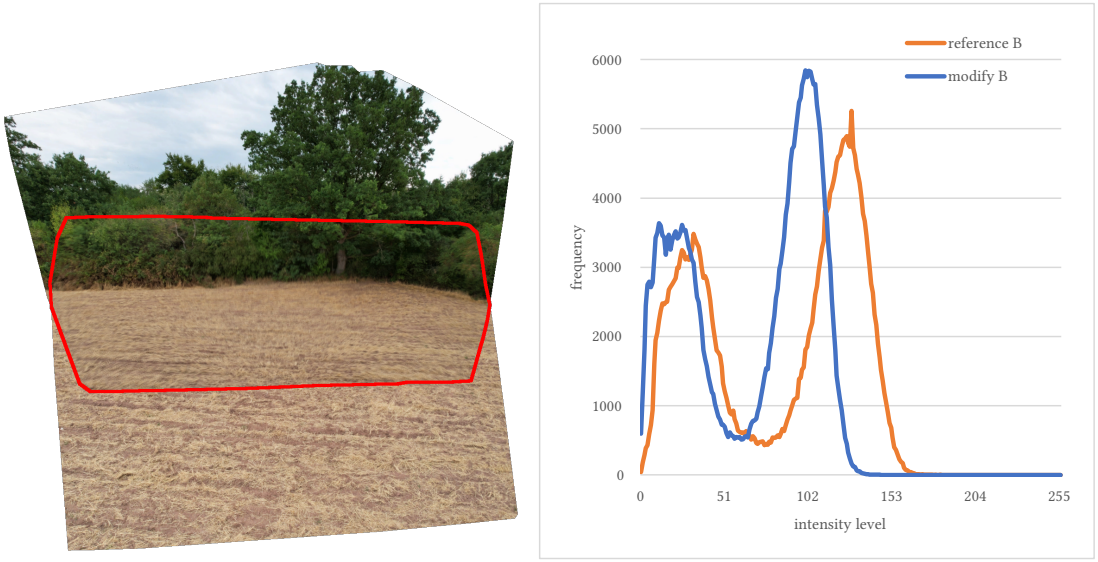
Figure 4: Multiple Cell Rendering. (a) Six morphing cells, consisting of three images each, are morphed to represent the same virtual camera position. While the morphable portion of a single morphing cell is too small to fill the whole viewport, drawing multiple cells can fill everything. (b) By drawing the colored wireframe of the morphing mesh on top, we visualize the render order with a heat color. The light yellow cell fits the virtual camera the most. The later a cell appears in the rendering ranking, the "colder" the wireframe color becomes.

2.5.3 Color Mapping

As can bee seen in fig. 4 from previous section, stitching together multiple morphing cells to fill the viewport can cause visible transitions between cells due to color inconsistencies. While it is possible to equalize the colors of the input images with preprocessing, using methods such as [32], a real-time adjustment can be desired in some circumstances: For example, when the user looks at a bright outside environment through a window, the inside of the room should appear darker while the outside should appear normal. When turning the view back to indoor, the virtual camera should adjust to the darker inside room and the outside should brighten up. Combined with HDR rendering, this is a common effect in today's video game graphic engines [33]. To achieve a similar effect in our renderer, a real-time color mapping technique based on histogram matching by Gonzales and Fittes [34] is introduced. To do this, we render the main cell first, which means that we inverse the render order from previous section. All succeeding cells adapt the color scheme of the pixels in the framebuffer which were set by previous cells. This happens in three steps per cell:

1. Histogram Acquisition In this first step, we compute the two histograms containing the pixel colors of the overlapping region of two cells. The *reference histogram* contains the colors of the overlapping region that are already present in the framebuffer. The *modify histogram* contains the colors of the overlapping region that the succeeding cell generates. Depending

on the number of color channels c and the number of states d per channel, a histogram consist of $c \cdot d$, e.g. $3 \cdot 256$ atomic counters. To acquire the histogram data, we draw the succeeding cell with a modified fragment stage compared to section 2.5.1. The modified fragment shader samples a copy of the old framebuffer. The copy is made right after the previous cell has finished drawing. The modified fragment shader differentiates two cases: If the old framebuffer at the target location already contains a color value, we add this already present color value to the reference histogram and the new color value to the modify histogram. In this case, we don't write the fragment to the framebuffer, and thus discard it. Otherwise, if the old framebuffer at the target location is empty, the new color value is set without any mapping. We use the, otherwise unused, alpha channel to differentiate between current and previously set pixels. E.g. if it is 0, the pixel is empty, if it is 0.5, the pixel still has to be color-mapped, and if it is 1, the pixel is final. As an optimization, it is not required to increment the histogram counters for the whole overlapping image region. As heavy use of atomic operations can be expensive, skipping a majority of fragments improves performance. For this purpose, a blue noise texture is sampled at the location of the fragment. The atomic counters are only incremented if this pseudo-random number lies below a threshold. We use a threshold of 50% in our implementation, thus only half of the overlap fragments are added to the histograms. Figure 5 gives an example of an overlapping region and its histograms.



(a) overlapping region

(b) blue channel histograms

Figure 5: Overlap Histogram. (a) Two cells with a slightly different color scheme are stitched and blended together. The red boundary marks the overlapping region from which the histograms are formed. (b) The frequency values of the blue channel histogram of the modify cell is slightly offset to the ones from the reference histogram. No brighter blue level than 145 was observed in the overlapping region for the modify image, which is an issue when we want to deduce a mapping for the sky color.

2. Mapping Computation In this second step, we compute the color mapping lookup table $f(i)$ for each color channel. $f(i)$ maps a single color channel from the *modify color space* to the *reference color space*. We define

$$G = \{\, g \in \mathbb{N} \mid 0 \leq g < d \,\} \tag{8}$$

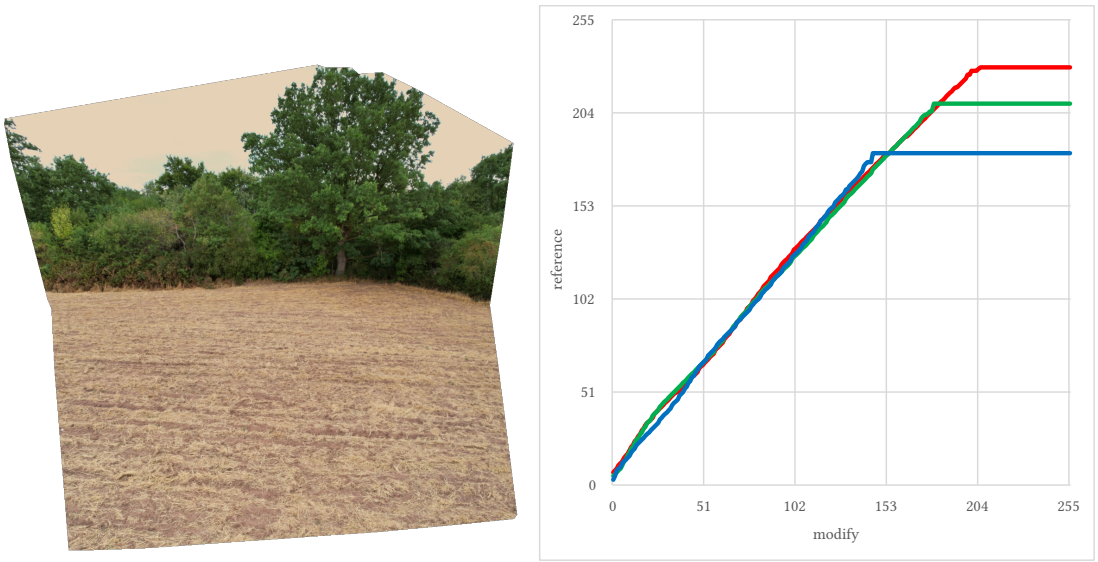
as the set of valid indices, or color intensity levels, to our histograms and mapping function $f(i)$. Following [34], we require the cumulative histograms $\hat{r}$ and $\hat{m}$, given the reference and modify histograms for a single channel r and m:

$$\hat{r}_g = \sum_{i=0}^{g} r_i, \quad \hat{m}_g = \sum_{i=0}^{g} m_i. \tag{9}$$

For each intensity level $i \in G$, we deduce the intensity level $j \in G$ for which $\hat{m}_i = \hat{r}_j$. As this is not always possible with discrete values, we can approximate this naïve color mapping for

a single color channel with:

$$f_{\text{naïve}}(i) = \min(\{\, j \in G \mid \hat{m}_i \leq \hat{r}_j \,\}). \tag{10}$$



(a) mapping result

(b) $f_{\text{naïve}}(i)$ for red, green, and blue

Figure 6: Naïve Mapping. (a) As the overlapping region only contains a limited range of colors, the mapping for the sky colors is inadequate. The cause of it can be seen in (b), where the mapping function clamps colors outside of the observed range to the last known mapping.

As can be seen in fig. 6 a flawed and disruptive mapping has been generated for the sky colors, while the mapping works properly for the ground elements. This artifact arises, when the range of colors observed in the overlapping region is smaller than the colors of the region to be mapped. Thus, in our example, if the overlapping region only shows grass and trees, the mapping technique does not have enough information to handle the blue sky color correctly. We define the *observed range* $[i_{\text{bot}}, i_{\text{top}}]$ of the modify histogram:

$$i_{\text{bot}} = \max(\{\, g \in G \mid \hat{m}_g = 0 \,\}) + 1 = \min(\{\, g \in G \mid \hat{m}_g \neq 0 \,\}) \tag{11}$$

$$i_{\text{top}} = \max(\{\, g \in G \mid \hat{m}_g \neq \max \hat{m} \,\}) + 1 = \min(\{\, g \in G \mid \hat{m}_g = \max \hat{m} \,\}). \tag{12}$$

Thus, i_{bot} is the lowest intensity value observed in the overlapping region in the modify image, while i_{top} is the highest. As $f_{\text{naïve}}(i)$ searches for the minimum index j where $\hat{m}_i \leq \hat{r}_j$, it is given that:

$$\forall i \in \{\, g \in G \mid g < i_{\text{bot}} \,\} f_{\text{naïve}}(i) = 0 \tag{13}$$

and

$$\forall i \in \{\, g \in G \mid i_{\text{top}} \leq g \,\} f_{\text{naïve}}(i) = f_{\text{naïve}}(i_{\text{top}}). \tag{14}$$

To counteract the flawed mapping outside of $[i_{\text{bot}}, i_{\text{top}}]$, we propose $f_{\text{strict}}(i)$ and $f_{\text{preserving}}(i)$. For the first method, we assume $f_{\text{naïve}}(i)$ to only offset each intensity value by a constant. This is the case, when the color difference is only a shift in exposure, thus in image brightness. Here, the mapping function inside the interval $[i_{\text{bot}}, i_{\text{top}}]$ would just be a line. For our improved mapping function, we *strictly* extrapolate that line to the unobserved range: Let

$$m = \frac{f_{\text{naïve}}(i_{\text{top}}) - f_{\text{naïve}}(i_{\text{bot}})}{i_{\text{top}} - i_{\text{bot}}} \quad \text{and} \quad \text{clamp}(x, a, b) = \min(\max(x, a), b) \tag{15}$$

in

$$f_{\text{strict}}(i) = \begin{cases} f_{\text{naïve}}(i), & \text{if } i_{\text{bot}} \leq i \leq i_{\text{top}} \\ \text{clamp}(\lfloor mi - mi_{\text{bot}} + f_{\text{naïve}}(i_{\text{bot}}) \rceil, 0, d-1), & \text{otherwise} \end{cases}. \tag{16}$$

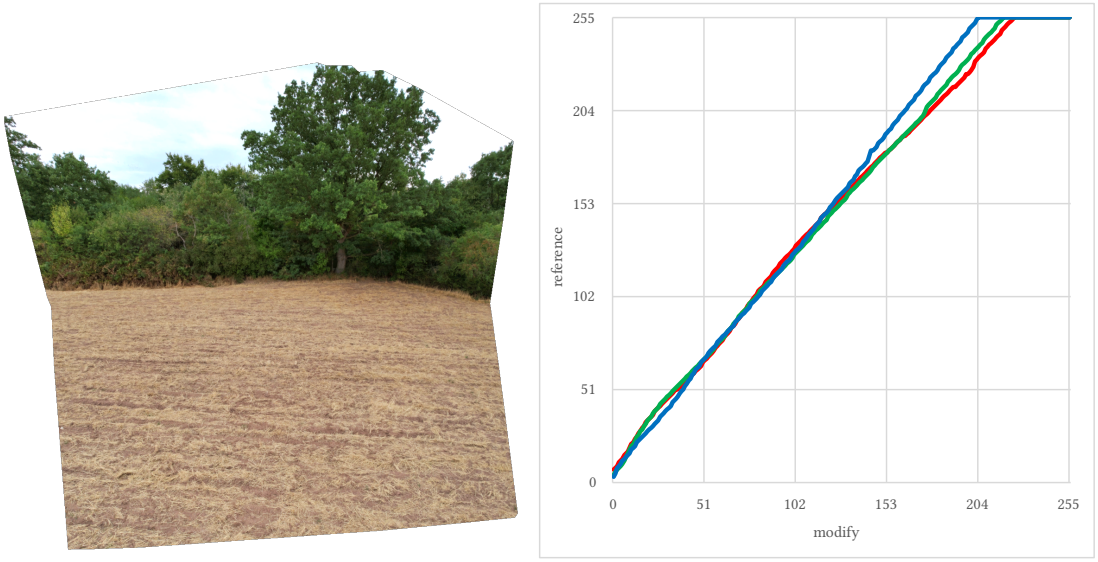
Figure 7 shows the result of strict mapping for our example. The mapping from the observed range is correctly extrapolated into the unknown range. This comes with a cost: The sky color becomes unconvincingly bright and cloud details disappear, because the mapping has to clamp multiple values to the maximum available intensity value.

For the second option, we want to *preserve* the color gradients of the unobserved color range, thus prevent clamping to the lowest or highest intensity. In cases where $f_{\text{strict}}(i)$ would cause clamping, we instead use linear interpolation between the closest known mapping and the identity mapping of the endpoints $f_{\text{preserving}}(0) = 0$ and $f_{\text{preserving}}(d-1) = d-1$: Let

$$\text{mix}(a, b, t) = (1-t)a + tb \tag{17}$$

in

$$f_{\text{preserving}}(i) = \begin{cases} 0 \leq i < i_{\text{bot}}, & \max\Big(f_{\text{strict}}(i), \Big\lfloor \text{mix}\Big(0, f_{\text{naïve}}(i_{\text{bot}}), \frac{i}{i_{\text{bot}}}\Big) \Big\rceil\Big) \\ i_{\text{bot}} \leq i \leq i_{\text{top}}, & f_{\text{naïve}}(i) \\ i_{\text{top}} < i < d, & \min\Big(f_{\text{strict}}(i), \Big\lfloor \text{mix}\Big(f_{\text{naïve}}(i_{\text{top}}), d-1, \frac{i-i_{\text{top}}}{d-1-i_{\text{top}}}\Big) \Big\rceil\Big) \end{cases}. \tag{18}$$



(a) mapping result

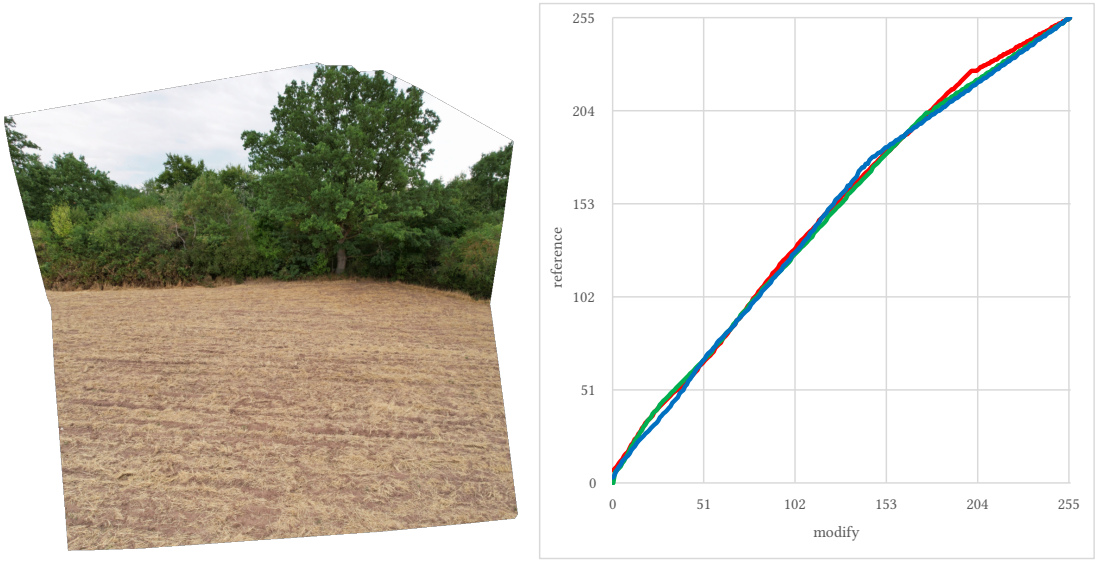
(b) $f_{\text{strict}}(i)$ for red, green, and blue

Figure 7: Strict Mapping. (a) The mapping for the overlapping region is correctly extrapolated to the sky colors. (b) Multiple values map to the maximum available intensity level.

Figure 8 shows the result of preserving mapping for our example. We continue where the naïve mapping ends and uniformly distribute all remaining color values. Preserving mapping is the method we recommend and use for the evaluation section later.

We perform the computation of the mapping function in a compute shader. Computations per channel do not depend on each other, therefore we invoke one thread per color channel to allow for some parallelization. Instead of computing the full cumulative histograms beforehand, we can have two accumulator variables in the same loop where the mapping is computed.

Depending on the dataset, it can be possible that there is no or only little overlap between the drawn cells. The number of acquired overlap samples is equal to $\hat{m}_{d-1} = \hat{r}_{d-1}$, thus the value of the accumulators after computing the naïve mapping. If this value falls below a threshold, we fall back to the identity mapping $f(i) = i$.



(a) mapping result

(b) $f_{\text{preserving}}(i)$ for red, green, and blue

Figure 8: Preserving mapping. (a) The color range observed in the overlapping region is mapped correctly to hide the cell transition, while the sky colors are kept close to the original color from the photograph. (b) For the unobserved color range, we generate a mapping by linearly interpolating from the last known mapping to a mapping of $f(255) = 255$.

3. Mapping Application In the last step, we apply the mapping computed in the previous step for all pixels that have not yet adopted their final value. If the alpha value of a pixel value is 1, we know the color is final. If it is 0.5, we apply the color mapping, set the alpha to 1, and write the new color to the framebuffer. If it is 0, thus empty, we leave it. Depending on what is better performing, we either use a compute shader to manipulate all pixels of the framebuffer or we redraw the morphing mesh with a modified fragment stage. In the second option, we narrow down the number of pixels touched, as a cell usually does not fill the whole screen.
Now, in case a switch of the main cell occurs, the colors of the rendered frame snap to adapt the color scheme of the new main cell. To smooth out this transition, and to imitate an eye slowly adjusting to a new level of brightness, we also apply a color mapping to the main cell. We start with the mapping table the cell had before it became the main cell, and slowly (e.g. in one second) interpolate to the identity mapping. Figure 9 shows a full rendering of the test scene with and without color mapping.

(a) without color mapping (b) with color mapping

Figure 9: Color Mapping. (a) Is the same as fig. 4. Transitions between cells are clearly visible. (b) Is the same scene and view, but with color mapping enabled. The color scheme of the center cell is propagated outwards and transitions become hardly noticeable.

2.5.4 VR Modifications and Optimizations

To create an enjoyable experience in VR, it is not sufficient to simply repeat the steps from sections 2.5.1 to 2.5.3 for a second view. Instead, we propose some modifications: First, we have to ensure that both views show the same cells in the same order. Otherwise, one view could be located at one side of a cell border, while the other view is on the opposite side. If the color scheme of the two cells differs, this can be immersion-breaking or even unpleasant for the user. Additionally, if the dataset is affected by preprocessing inaccuracies, an occurring cell switch can be visible. When each view has its own render order, the cell switch will be visible twice, once for each view. Instead of one transformation per eye, we use the center transformation between both eyes for determining a common order. The barycentric coordinates for morphing are then recomputed for each view individually. As an optimization, now, the color mapping of the view rendered first can also be used for mapping the second view. Additionally, to provide some guidance in places of a scene where data is not present in a full 360° surrounding, we draw a minimalistic floor at height level of the real floor. Figure 10 shows a stereo VR rendering created with our method.

Figure 10: Stereo VR Rendering. The color mapping in both image is consistent, because it is computed for the left eye only and then also used on the pixels of the right eye.

2.5.5 Combination with Conventional Rendering

Using our method as background scenery and combining it with conventional polygon-based 3D rendering for the foreground is straight forward: In section 2.5.1 we introduced a pseudo 3D position for each key point, based on the focal length of the source image as depth. These positions are then transformed with matrices well known from traditional rendering. Given a camera transformation, we first render the IBR scene. Next, we use the exact same camera and clip space transformation as for the IBR scene to transform the vertices of the polygon mesh into the clip space of our framebuffer. We must ensure that Z-buffering is disabled for IBR rendering and then enabled for polygon-based rendering. Figure 11 shows a synthetic IBR scene with a polygon mesh rendered inside of it.

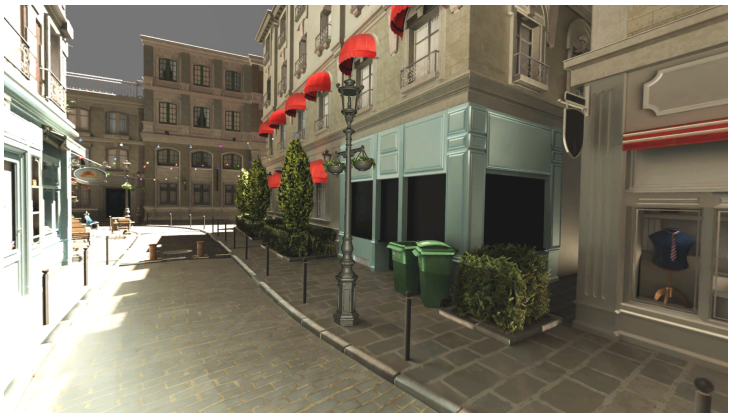


Figure 11: Polygon Mesh in IBR Scene. While we render the background with our IBR method, we render the lantern post using conventional polygon-based rasterization.

2.6 Results

In this section, we evaluate our rendering technique. We give an overview over our test scenes, present benchmarks, and outline limitations of our technique.

2.6.1 Test Scenes

Table 1 lists the set of test scenes we use for the evaluation. A rendering of each scene can be found in Figure 12. The set includes outdoor and indoor scenes, images of low and high resolution, synthetic and real creation processes, and a panorama dataset. We create synthetic scenes with the physically-based path tracer Cycles inside of Blender [35].

(a) farmland

(b) drone

(c) path

(d) living room

(e) bistro [36]

Figure 12: Test Scenes.

scene	type	degrees of translation $d-1$	data resolution $w \times h$
farmland	photography	3	1280×960
drone	photography	2	5472×3648
path	photography	3	5472×3648
living room	panorama	2	8192×4096
bistro	synthetic	1	2048×2048

Table 1: Test Scenes. Scenes consist either of real photographs, synthetic renderings, or multiple 360° panoramas. The dimension of the morphing cells defines the degrees of freedom in translation: lines (1), triangles (2), and tetrahedrons (3).

2.6.2 Performance

Table 2 lists performance measurements of our test scenes. All measurements were taken on an Intel i7-6700K CPU at 4.4 GHz with an AMD Radeon RX 6800 XT GPU and a screen resolution of 1920×1080. As can be seen, all measurements are well above the required rates for real time rendering. Enabling color mapping for the scenes that require it increases the frame time by approximately 25%. We speculate that performance could be improved further, if the implementation of our technique would be done with a lower level graphics API such as Vulkan [37]. Possibly even to the point that rendering on mobile devices becomes feasible. Additionally, these fast run times show that using our technique in combination with other rendering techniques is feasible, e.g. for dynamic sky maps.

scene	frame time (FPS)	colormap	#cells $\lvert C \rvert$	#triangles $\lvert T \rvert$	#vertices $\lvert V \rvert$
farmland	813µs (1230)	No	5	27862	13988
farmland	1048µs (954)	Yes	5	27862	13988
drone	1114µs (898)	No	12	161199	82274
drone	1571µs (637)	Yes	12	161199	82274
path	2765µs (637)	No	30	156560	78631
path	3752µs (267)	Yes	30	156560	78631
living room	659µs (1515)	No	1	26094	13120
bistro	800µs (1250)	No	3	57741	28977

Table 2: Performance Measurements. We measure the time it takes to render each image of fig. 12. Values are the median over 500 samples. Images from scenes that require color mapping are both measured with and without it. Depending on the field of view of the dataset images, each synthetic image requires a different number of cells for the viewport to be filled.

2.6.3 GPU Memory Demand

In this section, we give an overview over the GPU memory requirements of or image graph. The total memory requirements of a morphing cell with $d-1$ degrees of translation is given by the following quantities:

Triangles Each of the $|T|$ triangles of a morphing mesh consists of three indices to positions. As meshes, especially the ones for morphing panoramas, can have more positions than $2^{16} = 65536$, we use 4 byte integers per position index.

Positions Each of the $|P|$ positions of a morphing mesh has a 2D coordinate in each of the d member images. If the coordinates contain only whole numbers, 16 bit integers per coordinate are sufficient in most cases. In our case, some of our datasets contain positions with sub-pixel accuracy where we us 4 byte floating point numbers.

Textures To reduce bandwidth limitations, textures are stored on the GPU using BC1 (former DXT1) texture compression [38]. The technique takes a 4×4 block of pixels with, in our case, RGB data, where each pixel consists of $3 \cdot 8$ bits, and compresses it into 64 bits. Therefore, the compression ratio is

$$\frac{4 \cdot 4 \cdot 3 \cdot 8}{64}, \tag{19}$$

thus 6:1. Additionally, we use mipmapping to reduce aliasing artifacts. As each mipmap level is $\frac{1}{4}$ the size of the next bigger level, the n mipmap levels, including the first level, approximately contribute a factor of:

$$\lim_{n\to\infty} \sum_{i}^{n} \left(4^{-i}\right) = \frac{4}{3}. \tag{20}$$

Applying eqs. (19) and (20) and assuming that all textures of the cell have the same width w and height h, the total size in bytes of a morphing cell is

$$\underbrace{d \cdot |P| \cdot 2 \cdot 4\text{B}}_{\text{positions}} + \underbrace{|T| \cdot 3 \cdot 4\text{B}}_{\text{triangles}} + \underbrace{d \cdot \frac{4}{3} \cdot \frac{w \cdot h \cdot 3\text{B}}{6}}_{\text{textures}}. \tag{21}$$

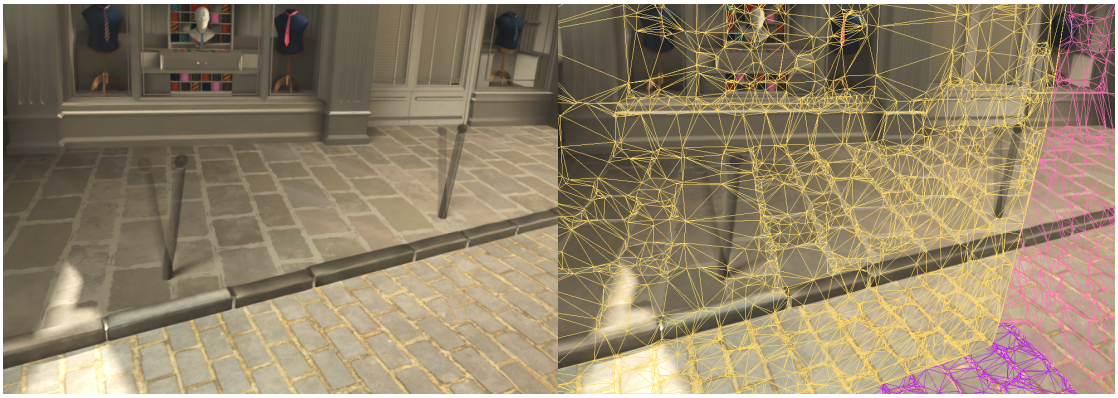
Table 3 applies this formula to the test frames from table 2. As can be seen, the size of the textures is the most significant factor, compared to the size of the morphing meshes, even with texture compression enabled.

scene	positions (%)	triangles (%)	textures (%)	sum (100%)
*	$d \cdot \lvert P \rvert \cdot 2 \cdot 4\text{B}$	$\lvert T \rvert \cdot 3 \cdot 4\text{B}$	$\lvert C \rvert \cdot d \cdot \frac{4}{3} \cdot \frac{w \cdot h \cdot 3\text{B}}{6}$	Σ
farmland	447KB (2.6%)	334KB (1.94%)	16.3MB (95.4%)	17.1MB
drone	1.97MB (0.307%)	1.93MB (0.3%)	638MB (99.3%)	642MB
path	2.51MB (0.157%)	1.87MB (0.117%)	1.59GB (99.7%)	1.6GB
living room	314KB (0.349%)	313KB (0.347%)	89.4MB (99.3%)	90.1MB
bistro	463KB (1.33%)	692KB (1.99%)	33.5MB (96.6%)	34.7MB

Table 3: Memory Requirements. We list the memory requirements for rendering each frame of fig. 12 and table 2. The size of the positions data (vertices), triangles data (position indices), and the textures are listed individually. For each entry, we show how much it represents in percentage of the total amount.

2.6.4 Limitations

The limitations of our IBR technique include ghosting artifacts, which appear where morphing-triangles that contain a parallax did not get filtered out. Figure 13 shows an example of such a case.



(a) ghosting artifact with bollards (b) wireframe

Figure 13: Ghosting Artifacts. The foreground bollards move at different pace than the background floor. The morphing mesh only accounts for the background, thus the blending results in ghosting artifacts. Additionally, our preprocessing struggles to create a detailed morphing mesh around the bollards. These larger triangles and a similar color of the bollards to the background complicate the parallax detection.

The quality of our multi cell rendering technique highly depends on the precision of the reconstructed extrinsic and intrinsic camera parameters. Extrinsic camera parameters refer to the camera pose, while the intrinsic camera parameters refer to camera hardware parameters, such as focal length, radial and tangential distortion, etc. Figure 14 shows how a flawed cam-

era calibration can have an effect with rendering. As the color mapping depends on a correct alignment, imprecisely calibrated datasets can also suffer from reduced quality color mapping.

Figure 14: Cell Alignment. Rotational cells with real images and flawed camera calibration can suffer from imprecise alignment.

In addition, our technique can only be used for static images, so it is not suitable for videos or motion other than camera motion. Furthermore, complicated parallaxes such as already shown in fig. 3 can only be approximated by extrapolating neighboring cells, which leads to gaps in the rendered image.

2.7 Conclusion

We have presented an IBR technique that uses a set of calibrated and densely feature-matched images of a scene as input. From the input data, we constructed an image graph that consists of morphing cells. We showed how to render these cells with today's graphics hardware. By rendering multiple cells on top of each other, we can handle simple parallaxes. To mitigate the problem of color inconsistencies within a dataset, we use a novel real time color mapping technique. The color mapping can imitate an eye slowly adjusting to new brightness levels, which is a common effect in today's polygon-based game engines. Our technique achieves frame times well above the required rates for real time rendering. In the future, we want to extend our technique with a more advanced parallax handling and fill gaps with real time inpaiting such as [39]. Additionally, we plan to extend on the hybrid rendering approach by reconstructing depth information to also mix IBR and polygon based assets in the foreground.

2.8 Bibliography

[1] Robert C Bolles, H Harlyn Baker, and David H Marimont. Epipolar-plane image analysis: An approach to determining structure from motion. *International journal of computer vision*, 1(1):7–55, 1987.

[2] Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M Seitz, and Richard Szeliski. Building rome in a day. *Communications of the ACM*, 54(10):105–112, 2011.

[3] Johannes L Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4104–4113, 2016.

[4] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, 2001.

[5] Shenchang Eric Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 29–38, 1995.

[6] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 279–288, 1993.

[7] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *ACM SIGGRAPH computer graphics*, 26(2):35–42, 1992.

[8] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46, 1995.

[9] Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54, 1996.

[10] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, 1996.

[11] Steven M Seitz and Charles R Dyer. View morphing. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 21–30, 1996.

[12] Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Workshop on Rendering Techniques*, pages 105–116. Springer, 1998.

[13] Gaurav Chaurasia, Sylvain Duchene, Olga Sorkine-Hornung, and George Drettakis. Depth synthesis and local warps for plausible image-based navigation. *ACM Transactions on Graphics (TOG)*, 32(3):1–12, 2013.

[14] Tobias Bertel and Christian Richardt. Megaparallax: 360° panoramas with motion parallax. In *ACM SIGGRAPH 2018 Posters*, pages 1–2. 2018.

[15] Tobias Bertel, Neill DF Campbell, and Christian Richardt. Megaparallax: Casual 360° panoramas with motion parallax. *IEEE transactions on visualization and computer graphics*, 25(5):1828–1835, 2019.

[16] Jens Ogniewski. High-quality real-time depth-image-based-rendering. In *SIGRAD 2017, August 17-18, 2017 Norrköping, Sweden*, number 143, pages 1–8. Linköping University Electronic Press, 2017.

[17] Daniele Bonatto, Sarah Fachada, Ségolène Rogge, Adrian Munteanu, and Gauthier Lafruit. Real-time depth video-based rendering for 6-dof hmd navigation and light field displays. *IEEE access*, 9:146868–146887, 2021.

[18] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (TOG)*, 37(6):1–15, 2018.

[19] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *arXiv preprint arXiv:1805.09817*, 2018.

[20] Ben Mildenhall, Pratul P Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*, 38(4):1–14, 2019.

[21] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. Deepview: View synthesis with learned gradient descent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2367–2376, 2019.

[22] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pages 405–421. Springer, 2020.

[23] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *Advances in Neural Information Processing Systems*, 33:15651–15663, 2020.

[24] Darius Rückert, Linus Franke, and Marc Stamminger. Adop: Approximate differentiable one-pixel point rendering. *arXiv preprint arXiv:2110.06635*, 2021.

[25] Simon Seibt, Bartosz Von Rymon Lipinski, and Marc Erich Latoschik. Dense feature matching based on homographic decomposition. *IEEE Access*, 10:21236–21249, 2022.

[26] Boris Delaunay et al. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.

[27] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[28] Georgios D Evangelidis and Emmanouil Z Psarakis. Parametric image alignment using enhanced correlation coefficient maximization. *IEEE transactions on pattern analysis and machine intelligence*, 30(10):1858–1865, 2008.

[29] Manuel Caroli, Pedro MM de Castro, Sébastien Loriot, Olivier Rouiller, Monique Teillaud, and Camille Wormser. Robust and efficient delaunay triangulations of points on or close to a sphere. In *International Symposium on Experimental Algorithms*, pages 462–473. Springer, 2010.

[30] Khronos Group. smoothstep. https://registry.khronos.org/OpenGL-Refpages/gl4/html/smoothstep.xhtml. Accessed: 2022-10-11.

[31] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[32] Qi-Chong Tian and Laurent D Cohen. Histogram-based color transfer for image stitching. *Journal of Imaging*, 3(3):38, 2017.

[33] Crytek GmbH. Tutorial - Environment Editor HDR and Filters. https://docs.cryengine.com/display/CEMANUAL/Tutorial+-+Environment+Editor+Part+4+-+HDR+and+Filters. Accessed: 2022-09-29.

[34] Rafael C Gonzales and BA Fittes. Gray-level transformations for interactive image enhancement. *Mechanism and Machine Theory*, 12(1):111–122, 1977.

[35] Blender Community. Blender. http://www.blender.org, 2022.

[36] Amazon Lumberyard. Amazon lumberyard bistro, open research content archive (orca). http://developer.nvidia.com/orca/amazon-lumberyard-bistro, July 2017.

[37] Khronos Group. Vulkan api. https://vulkan.org/, 2022.

[38] Pat Brown, Ian Stewart, Nicholas Haemel, Acorn Pooley, Antti Rasmus, and Musawir Shah. EXT_texture_compression_s3tc. https://registry.khronos.org/OpenGL/extensions/EXT/EXT_texture_compression_s3tc.txt. Accessed: 2022-10-11.

[39] Jan Herling and Wolfgang Broll. High-quality real-time video inpainting with pixmix. *IEEE Transactions on Visualization and Computer Graphics*, 20(6):866–879, 2014.

3 GPU Accelerated Optimization of Image-Morphing Meshes for Visual Coherence

3.1 Abstract

We propose a method for genetically optimizing a given triangle mesh that morphs image elements between two or more images for visual quality. We parallelize the *enhanced correlation coefficient* image alignment error metric and utilize the rasterization capabilities of graphics hardware to evaluate mutations on the vertices of the triangles. We show that morphing with such an optimized mesh will result in sharper images and less ghosting artifacts in the context of image-based rendering. We demonstrate and evaluate our approach on several examples from the field of image-based rendering.

3.2 Introduction

Image Morphing is usually referred to as the process of transforming image elements from at least one source image to a target image, while simultaneously blending the colors. Most prominently this is used in cinematography and image-based rendering. Other use-cases include video editing where in-between frames are synthesized from two key frames, e.g. to increase the frame rate of a shot or slow down the motion. One method for transforming the image elements is to use *morphing meshes*. A morphing mesh consists of vertices defined by point feature matches between the images. After matching, these vertices are triangulated to a mesh, such that the triangulation is equal in all images. If the images are reasonably similar, interpolating the feature point positions, thus transforming the triangles, can result in a seamless transition when combined with blending. However, today's feature matching techniques do optimize for highest matching quality at a triangle vertex, thus not inside the triangles. Therefore, in this work, we propose a method for optimizing a given morphing mesh for visual coherence by exploiting the rasterization capabilities of graphics hardware. Instead of employing a pure *general purpose computation on graphics processing unit* (GPGPU) based approach with APIs such as CUDA [1], we make direct use of the rasterization capabilities and hardware of a GPU in an OpenGL environment. In detail, we make the following contributions:

- We parallelize the computation of the *enhanced correlation coefficient* (ECC) [2] image alignment error norm by splitting its computation into multiple sums.
- We suggest a genetic algorithm that optimizes a morphing mesh for that error norm.

- We show how to efficiently use a GPU to evaluate numerous mutations with a single draw call.
- We show that our technique can improve the visual quality when morphing with examples from image-based rendering.

3.3 Related Work

Several related methods exist that optimize correspondences between images for morphing by measuring different choices of error or cost metrics. The method by Gao and Sederberg [3] measures the sum of the euclidean distance between color vectors. Their optimization algorithm is already closely related to our algorithm in section 3.7. Methods such as by Wu and Liu [4] extend that by combining the euclidean distance of colors with the euclidean distance of color gradients. Mhajan et al. [5] noted that e.g. a doubling of both the source and target color values would undesirably result in a multiple of the original error metric. Thus, they propose a normalized version of this metric by adding the standard deviation as a divisor. Other methods such as by Liao et al. [6] compose an error metric by combining different constraints to a energy function, including the structural similarity index (SSIM) introduced by Wang et al. [7].

With the introduction of programmable graphics hardware, GPU based feature matching techniques have emerged. Today it is common to speed up feature matching techniques with GPGPU. For example, Sinha et al. [8] make use of programmable fragment stages to implement SIFT in an OpenGL graphics environment. More modern approaches make use of direct GPGPU APIs like CUDA, e.g. [6]. However, we could not find any existing technique that makes direct use of a hardware accelerated rendering pipeline with a vertex and fragment stage to solve a feature matching problem.

3.4 Input Dataset

Our technique takes a triangle morphing mesh as input. The morphing mesh describes how image elements are transformed between two or more images. Each vertex of the mesh has a position per image. The positions are then triangulated equally in each image space. By interpolating between the vertex positions of the mesh and simultaneously blending between the color values of the images, an in-between view can be approximated. We assume that the input morphing mesh does not degenerate, i.e. points do not move over a triangle edge during morphing. Most commonly, the morphing meshes positions are generated using a point-based

feature matching technique. As a result, such techniques do not optimize for the best alignment of the triangles that connect the feature matches.

3.5 Error Norm

To measure the quality of the morphing mesh alignment, the *enhanced correlation coefficient* (ECC) by Evangelidis and Psarakis [2] is used. We choose this error norm, because it is an established way for quantifying the quality of image alignment [9]. Additionally, its computation is easily adaptable for GPUs, which is shown in this section. To quantify the quality of the overall overlap, we require the source vector of pixel intensities $\vec{x}$ and target vector of pixel intensities $\vec{y}$. Each intensity value $\vec{x}_i$ overlays with the value $\vec{y}_i$. The ECC is defined as:

$$E_{\text{ECC}}(\vec{x}, \vec{y}) = \frac{(\vec{x} - \bar{x}) \cdot (\vec{y} - \bar{y})^T}{||\vec{x} - \bar{x}|| \cdot ||\vec{y} - \bar{y}||}, \tag{1}$$

where $\bar{x}$ and $\bar{y}$ are the average intensity levels of each vector. In detail, if the ECC is 1, each of the pixel values are exactly the same. If it is 0, the intensity levels do not correlate. If the coefficient is -1, the images exactly counter-correlate, thus the source image is the negative of the target image. As we optimize an already present morphing mesh, in our case, the initial ECC is expected to be positive. Expanding this concept to color channels $x_r x_g x_b$ is trivial: e.g. $\vec{x} = \begin{bmatrix} \vec{x_r}_1 & \dots & \vec{x_r}_n & \vec{x_g}_1 & \dots & \vec{x_g}_n & \vec{x_b}_1 & \dots & \vec{x_b}_n \end{bmatrix}$. If we want to evaluate the alignment of a source to multiple targets $\vec{u}$, $\vec{v}$ and $\vec{w}$ at once, the target values are adjoined to a single vector and the source values are repeated, thus

$$\vec{x} = \begin{bmatrix} \vec{x}_1 & \dots & \vec{x}_n & \vec{x}_1 & \dots & \vec{x}_n & \vec{x}_1 & \dots & \vec{x}_n \end{bmatrix}, \tag{2}$$

$$\vec{y} = \begin{bmatrix} \vec{u}_1 & \dots & \vec{u}_n & \vec{v}_1 & \dots & \vec{v}_n & \vec{w}_1 & \dots & \vec{w}_n \end{bmatrix}. \tag{3}$$

As we want to parallelize the computation of the error norm, it is split into multiple sums:

$$E_{\text{ECC}}(\vec{x}, \vec{y}) = \frac{(\vec{x} - \bar{x}) \cdot (\vec{y} - \bar{y})^T}{||\vec{x} - \bar{x}|| \cdot ||\vec{y} - \bar{y}||} = \frac{\sum_i^n (\vec{x}_i \vec{y}_i - \vec{x}_i \bar{y} - \vec{y}_i \bar{x} + \bar{x}\bar{y})}{\sqrt{\sum_i^n (\vec{x}_i - \bar{x})^2 \cdot \sum_i^n (\vec{y}_i - \bar{y})^2}} = \tag{4}$$

$$= \frac{\sum_i^n (\vec{x}_i \vec{y}_i) - \sum_i^n (\vec{x}_i \bar{y}) - \sum_i^n (\vec{y}_i \bar{x}) + n\bar{x}\bar{y}}{\sqrt{\sum_i^n (\vec{x}_i^2 - 2\vec{x}_i \bar{x} + \bar{x}^2) \cdot \sum_i^n (\vec{y}_i^2 - 2\vec{y}_i \bar{y} + \bar{y}^2)}} = \tag{5}$$

$$= \frac{\overbrace{\sum_i^n (\vec{x}_i \vec{y}_i)}^{\widehat{xy}} - \bar{y} \overbrace{\sum_i^n (\vec{x}_i)}^{\hat{x}} - \bar{x} \overbrace{\sum_i^n (\vec{y}_i)}^{\hat{y}} + n\bar{x}\bar{y}}{\sqrt{(\underbrace{\sum_i^n \left(\vec{x}_i^2\right)}_{\widehat{xx}} - 2\bar{x} \underbrace{\sum_i^n (\vec{x}_i)}_{\hat{x}} + n\bar{x}^2) \cdot (\underbrace{\sum_i^n \left(\vec{y}_i^2\right)}_{\widehat{yy}} - 2\bar{y} \underbrace{\sum_i^n (\vec{y}_i)}_{\hat{y}} + n\bar{y}^2)}} = \tag{6}$$

$$= \frac{\widehat{xy} - \frac{2\hat{x}\hat{y}}{n} + \frac{\hat{x}\hat{y}}{n}}{\sqrt{\left(\widehat{xx} - \frac{2\hat{x}\hat{x}}{n} + \frac{\hat{x}\hat{x}}{n}\right) \cdot \left(\widehat{yy} - \frac{2\hat{y}\hat{y}}{n} + \frac{\hat{y}\hat{y}}{n}\right)}} = \frac{\widehat{xy} - \frac{\hat{x}\hat{y}}{n}}{\sqrt{\left(\widehat{xx} - \frac{\hat{x}\hat{x}}{n}\right) \cdot \left(\widehat{yy} - \frac{\hat{y}\hat{y}}{n}\right)}} = \tag{7}$$

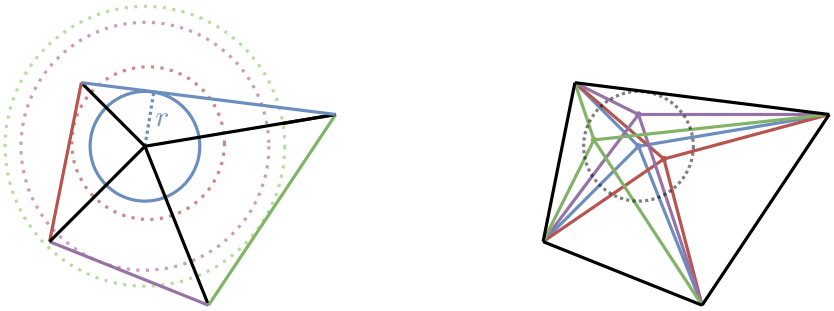
$$= \frac{n\widehat{xy} - \hat{x}\hat{y}}{\sqrt{n}\sqrt{n}\sqrt{\left(\widehat{xx} - \frac{\hat{x}\hat{x}}{n}\right) \cdot \left(\widehat{yy} - \frac{\hat{y}\hat{y}}{n}\right)}} = \frac{n\widehat{xy} - \hat{x}\hat{y}}{\sqrt{(n\widehat{xx} - \hat{x}\hat{x}) \cdot (n\widehat{yy} - \hat{y}\hat{y})}}. \tag{8}$$

Thus we separate the ECC computation into the tuple $(\hat{x}, \hat{y}, \widehat{xx}, \widehat{yy}, \widehat{xy}, n)$, consisting of 6 accumulators. We call this tuple the *preECC*. This approach is inspired by the recommendations by Chan et al. [10] for combining two samples when calculating the overall variance. Now, with this analysis, we are able to fully parallelize the ECC computation by having multiple separate preECCs and combining them in a final, single threaded step.

3.6 Mutation and Evaluation

Using our, now parallelizable, ECC computation, we want to evaluate whether a change of position of a feature point improves the alignment of the morphing triangles that are affected by that point. For this, the standard rendering pipeline of, in our case, OpenGL with a Vertex and Fragment stage is used. Instead of writing pixels to a framebuffer, we are only interested in the quality of the overlap, thus the ECC. The framebuffer size is set to the size of the input images. We did not observe any benefits of using multi-sampling, thus evaluating the image alignment at more points than the original images, which comes with a run time cost. We find all triangles T_P of the morphing mesh that use the feature point P that should be optimized, and draw them M times using *GPU instancing*. GPU instancing is a standard feature in graphic APIs and allows for rendering of multiple instances of the same object, thus an element buffer, very efficiently with a single draw call [11]. In the vertex stage, we receive the source and target feature point position as vertex attributes. If the currently processed vertex is the one we want to mutate, we offset its position to a pseudo random position inside a circle with radius r around the current position. The random value is based on the instance ID as a seed combined

with a global seed passed from the CPU as a uniform variable. Additionally, we ensure that the position for instance ID 0 remains unchanged to also consider the original position in our evaluation. The value of r prevents the mesh to degenerate, thus P moving to a position outside of the triangle area spanned by T_P. Therefore r must be the minimum distance between p and its opposing edges. This is also shown in fig. 15. As the fragment position, we pass over the average of the source and target position, which corresponds to a morph directly in between both images.



(a) radius computation (b) 4 instances

Figure 15: Maximum Mutation Radius. (a) The center feature point is connected to multiple triangles. The closest opposing edge (blue) defines the maximum mutation radius r. All the other edges are further away. (b) We evaluate 3 mutations and the original position of the central point.

In the fragment stage, we sample both images using the feature point's positions passed from the vertex shader as texture coordinates and compute the preECC of the current fragment by accumulating each color channel. We create a global buffer of size $M \cdot 6$ for the 6 preECC values per instance. The 6 local preECC values for the current fragment are then added to the 6 global counters responsible for all the fragments of the instance ID. As GPUs process several fragments in parallel, we have to use a synchronized atomic add operation for this step. Due to round off errors, IEEE 754 floating-point numbers do not always hold the associative laws of algebra [12]. Hence, when a GPUs internal scheduler changes the order of computation, thus the order fragments are added to the global counters, the resulting global preECC counters can have different values for the exact same input. To counteract this, we use (unsigned) integer arithmetic instead. While some image formats would require the use of quantization, all of our test images already come with a color depth d of 1 byte or $d = 256$ integral states. When evaluating a mutation of e.g. a point that is connected to large triangles, thus affects many pixels, the counters $\widehat{xx}$, $\widehat{yy}$, and $\widehat{xy}$ threaten to overflow for 32 bit arithmetic. Given the

maximum color value $d-1$, the number of color channels c, the number of target images t and the number of fragments given by the total area A of the triangles that are connect to the current point, the worst case maximum value $V \in \mathbb{N}$ for the problematic atomic counters $\widehat{xx}$, $\widehat{yy}$, and $\widehat{xy}$ is

$$V = (d-1)^2 \cdot c \cdot t \cdot A. \tag{9}$$

If V does not fit into the used integer size e.g. 32 bit, we shift right, thus scale down, each value by $\max(0, \lceil \log_2(V) \rceil - 32)$ rounding bits before adding it to the global preECC counters. When computing the final ECC, we reverse this rounding operation in floating point arithmetic.
To turn the preECC into the final ECC, we invoke a compute shader with M threads. The instance ID with the highest ECC represents the best mutation. We now can mutate a feature point position M times and find the best mutation in just two draw calls: one for drawing and one for the compute shader.

3.7 Optimization Algorithm

For optimizing a whole mesh, we apply the algorithm in fig. 16. To optimize a feature point's position in every target mesh, we rotate the mutation target around until a threshold is reached. To constrain the mutation for large triangles, it can be helpful to introduce a global maximum radius R which is reduced after each full cycle by a decay factor D. Depending on the graphics hardware in use, it can be beneficial to use a constant number of instances M tailored to the hardware (e.g. the number of compute units) or to scale it dynamically based on the area to cover for the current point, thus $M' = M\frac{\pi r^2}{\pi R^2}$.
In the actual implementation, we compute the triangles that are connected to each point beforehand. Additionally, we keep all required geometry data on the GPU at all times and only update it with the improved positions when required.

Input : The morphing mesh feature point positions
$P = \{P_0, P_1, [\text{additional morphing targets}]\}$
$|P_0| = |P_1| = |P_i|$, $P_{i_j} \in \mathbb{R}^2$,
Morphing mesh triangles T. Each triangle consists of three indices.
Minimum percentage improvement threshold t, e.g. 1%
Output: The optimized feature point positions

$\bar{E}' \leftarrow -1$
repeat *until improvement falls below threshold*
  $\bar{E} \leftarrow \bar{E}'$
  $\bar{E}' \leftarrow 0$
  for $i \leftarrow 0$ **to** $|P| - 1$ **do** *each target*
    for $j \leftarrow 0$ **to** $|P_i| - 1$ **do** *each point*
      $T_{\text{point}} \leftarrow \{\, t \in T \mid j \in t \,\}$ *find point triangles*
      $s \leftarrow$ *new random seed*
      $r \leftarrow$ *maximum radius like in fig. 15*
      $m, E \leftarrow$ *apply section 3.6, given* (i, P, T_{point}, s, r)*, return best ID and ECC*
      $\bar{E}' \leftarrow \bar{E}' + E$ *sum ECCs for average*
      if $m \neq 0$ **then**
        recompute mutation on CPU and add it to the point
        $P_{i_j} \leftarrow P_{i_j} + \text{mutation}(m, s)$
      end
    end
  end
  $\bar{E}' \leftarrow \frac{\bar{E}'}{|P| \cdot |P_i|}$ *convert ECC sum to average ECC*
until $\frac{\bar{E}'}{\bar{E}} - 1 \leq t$
return P

Figure 16: Algorithm for Optimizing a Morphing Mesh. We loop over each target image space and optimize each position individually, until the relative improvement falls below a threshold.

3.8 Results

In this section, we evaluate our optimization technique. We search for good meta parameters, provide benchmarks and show visual results. All measurements were taken on an Intel i7-6700K CPU at 4.4 GHz with an AMD Radeon RX 6800 XT GPU.

3.8.1 Meta Parameter Optimization

In this section, we evaluate what combination of meta parameters M, R, and D from section 3.7 optimizes a mesh well in terms of run time performance and final quality. We choose a set of

possible parameters and measure each combination.

Figure 17 plots all obtained measurements. While most configurations eventually settle at a similar final ECC value, it can be seen that combinations without a maximum radius (markers with red edges), and thus also without radius decay, either take significantly longer to reach the final ECC or terminate early. Next, combinations with a higher number of instances (marker shape with more corners) tend to take longer than configurations with 32 or 64 instances per draw call. Additionally, it can be seen that introducing some radius decay (blue and orange inner marker color) has a benefit in both final quality and run time. Finally, most markers of the same kind scatter at a similar area. Thus, our technique produces a similar output given the same input, despite using different random seeds each time.

In conclusion, for all subsequent evaluations we use an instance count of $M = 64$, a global maximum radius of $R = 5$, and a radius decay of $D = 0.5$.

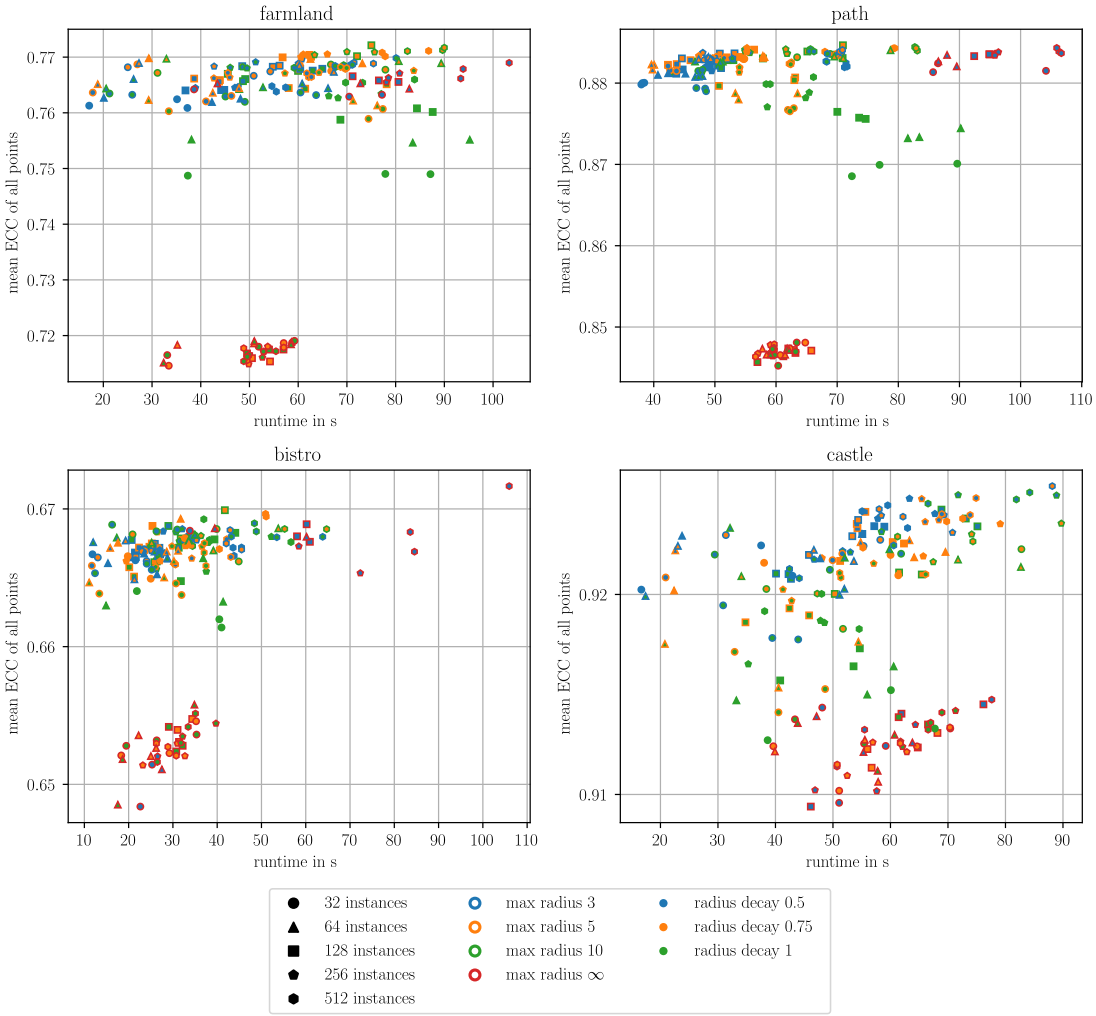
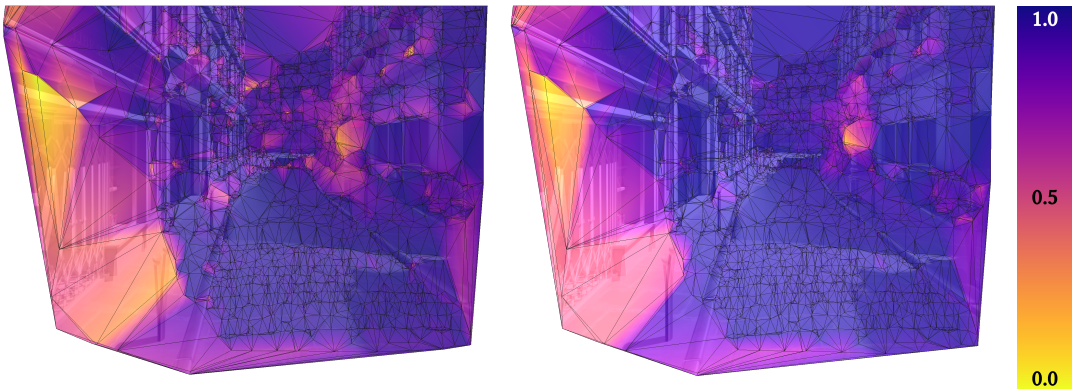


Figure 17: Meta Parameter Measurements. For each of four test scenes, we evaluate every combination of a set of meta parameters and plot computation time in the horizontal axis (lower is better) against the final optimized average ECC over all vertices in the vertical axis (higher is better). We set t to 0.005, thus when the ECC improvement of one step is less than 0.5%, we consider the dataset to have converged and terminate the optimization. The marker shape represents the number of instances M, which is scaled dynamically like described in section 3.7. The marker edge color represents the initial global maximum mutation radius R. The inner marker color represents the radius decay D factor after each full cycle. We measure and plot each combination with each dataset three times.

3.8.2 Visual Results and Benchmarks

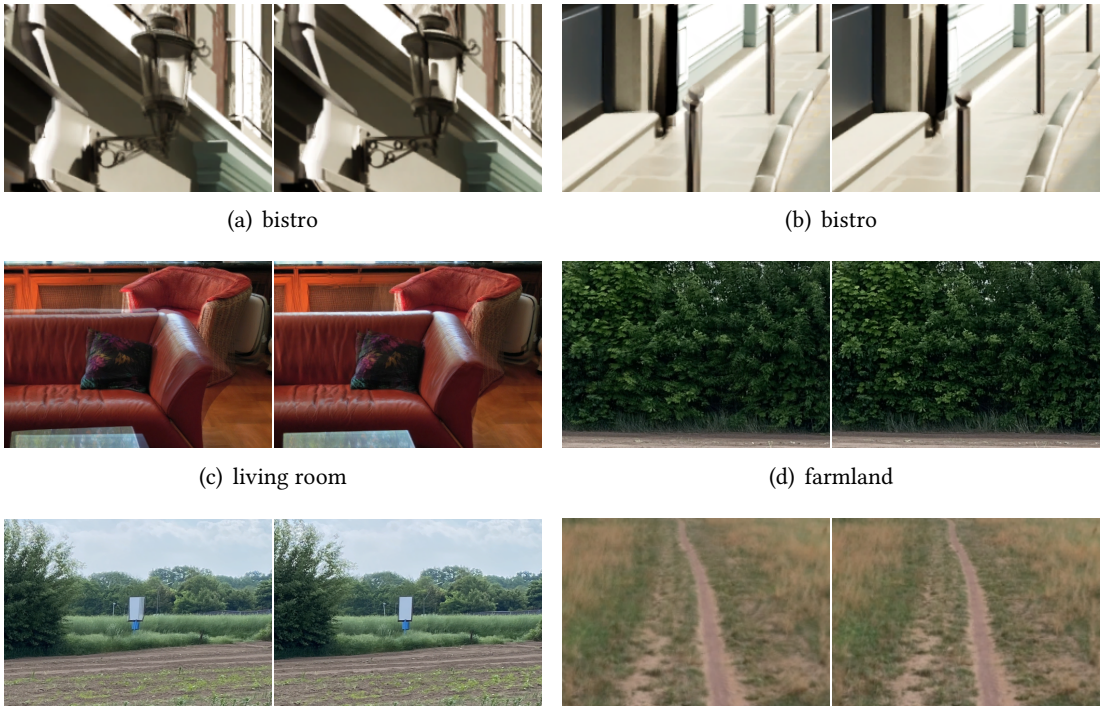
After finding good meta parameters for our technique in the previous section, in this section, we compare between optimized and unoptimized meshes. Figure 18 shows a visualization of a mesh before and after the optimization process. As can be seen, our technique can increase the correlation, thus the alignment, for a majority of vertices. However, this is not the case for areas where the input morphing mesh is already too deficient. Some areas, where image elements are only present in some of the morphing targets, e.g. because of parallaxes, or where the input geometry is too sparse, can not be optimized.



(a) initial mean ECC of 0.625237 (b) optimized mean ECC of 0.666981

Figure 18: Before and After ECC Visualization. (a) Shows the central morph between two images, overlaid by the used morphing mesh and an ECC heat map. The "hotter" the color surrounding a vertex, the worse the triangles connected to the vertex align between the images. A bad alignment means a low correlation, thus a low ECC value. (b) Shows this visualization after the optimization process. The heat map only maps ECC values between 0 and 1 because there is are no counter-correlating points present.

Figure 19 presents hand-picked areas of our test scenes, which show that our technique can reduce or completely remove ghosting artifacts, align detailed features such as vegetation, and improve the overall sharpness of the morphed image. Table 4 lists benchmarks to these examples.



(a) bistro (b) bistro

(c) living room (d) farmland

(e) farmland (f) path

Figure 19: Before and After Comparison. Selected areas of our test datasets are shown before and after applying our technique. Each image shows the center morph between 2 (a, b, c), 3 (d, e) and 4 (f) images.

Scene	#images	resolution	#points	initial ECC	final ECC	run time
farmland	3	1280×960	2544	0.678	0.765	18.8s
path	4	1280×853	2827	0.790	0.882	39.6s
bistro	4	2048×2048	4158	0.648	0.667	15.8s
castle	2	1280×853	13087	0.896	0.922	22.5s
living room	2	2048×2048	1505	0.745	0.759	25.1s

Table 4: Additional Benchmarks. We optimize each of our test scenes and measure the time it takes to optimize the mean initial ECC of all points to the a final value.

3.9 Conclusion

We proposed a GPU accelerated method for genetically optimizing the alignment of an input morphing mesh. As an optimization criterion, we adapted the *enhanced correlation coefficient* computation to be computed in parallel on a GPU. By using GPU instancing, a point is efficiently mutated several times with a single draw call and evaluated using a compute shader. We showed that morphing with such an optimized mesh will result in sharper images and less

ghosting artifacts. In the future, we want to explore methods for adding and subsequently optimizing geometry where the morphing error remains high after the initial optimization.

3.10 Bibliography

[1] NVIDIA Corporation. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/index.html. Accessed: 2022-12-1.

[2] Georgios D Evangelidis and Emmanouil Z Psarakis. Parametric image alignment using enhanced correlation coefficient maximization. *IEEE transactions on pattern analysis and machine intelligence*, 30(10):1858–1865, 2008.

[3] Peisheng Gao and Thomas W Sederberg. A work minimization approach to image morphing. 1998.

[4] Enhua Wu and Feitong Liu. Robust image metamorphosis immune from ghost and blur. *The visual computer*, 29(4):311–321, 2013.

[5] Dhruv Mahajan, Fu-Chung Huang, Wojciech Matusik, Ravi Ramamoorthi, and Peter Belhumeur. Moving gradients: a path-based method for plausible image interpolation. *ACM Transactions on Graphics (TOG)*, 28(3):1–11, 2009.

[6] Jing Liao, Rodolfo S Lima, Diego Nehab, Hugues Hoppe, Pedro V Sander, and Jinhui Yu. Automating image morphing using structural similarity on a halfway domain. *ACM Transactions on Graphics (TOG)*, 33(5):1–12, 2014.

[7] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[8] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22(1):207–217, 2011.

[9] Open Source Computer Vision Library (OpenCV). computeECC. https://docs.opencv.org/4.6.0/dc/d6b/group__video__track.html#gae94247c0014ff6497a3f85e60eab0a66. Accessed: 2022-12-1.

[10] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[11] Michael Gold. EXT_draw_instanced. https://registry.khronos.org/OpenGL/extensions/EXT/EXT_draw_instanced.txt. Accessed: 2022-10-31.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

4 Additional Results

4.1 Software Architecture

In this section we outline noteworthy aspects of the developed software architecture.

4.1.1 Class Hierarchy

To introduce a reusable code base for multiple different rendering applications, we construct a hierarchy of classes. Figure 20 visualizes part of the class hierarchy we implemented. The topmost class, thus the ancestor of all our rendering applications is *GLFWApp*. It encapsulates functionality like window creation using the multi-platform *GLFW* API [1], or user interface rendering with the *Dear ImGUI* [2] library. The class has several virtual methods that can be overridden by child classes to provide functionality like input handling (onKey, onCursorPosition), guidance when to draw what (onDraw, onDrawUI), or events such as window re-scaling (onWindowResize). A child class of GLFWApp is *XRApp*. Apps that have VR functionality inherit from this class. XRApp abstracts the communication with a VR or AR device using the *OpenXR* API [3]. It provides additional virtual methods for drawing each eye of the VR device (onDrawEye) or to the screen (onDrawScreen). If it is not overridden, onDrawScreen just replicates the content shown on the VR device by copying the framebuffers. Additionally, it keeps track of the VR devices position relative to a VR coordinate system (getCam2VR) and VR controller input (getActionValue).
The class *Graph* contains a full IBMPP dataset. Besides keeping track of the nodes (class *Node*) and links (class *Link*) of the image graph, it contains member variables that are local to the scene, e.g. the virtual camera transformation. When a user switches between multiple scenes, the virtual camera position of the scene remains at the location where it was left. Finally, Graph, Node, and Link all have GL counterparts (e.g. *GLLink*), which encapsulate the respective handles to the OpenGL objects for rendering, instead of CPU data. More information about the encapsulation of OpenGL resources can be found in the next section.

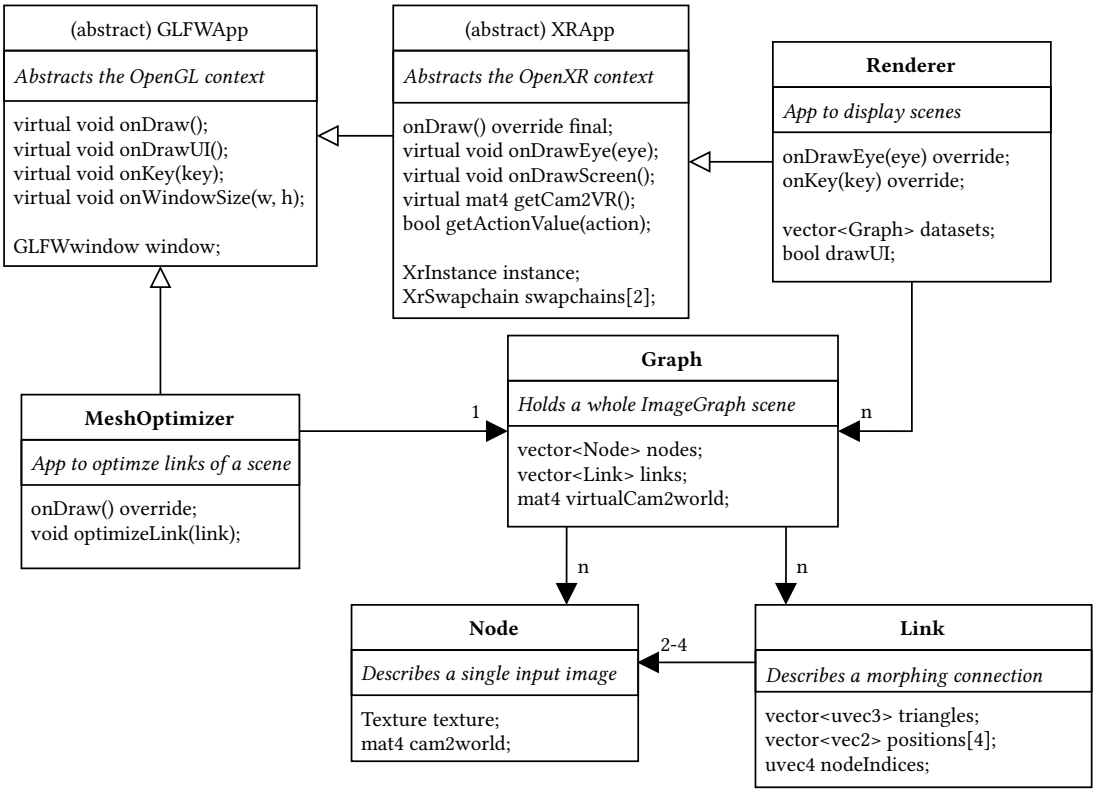


Figure 20: Class Hierarchy and Relations Overview. An empty arrow tip indicates a class inheritance, a filled arrow an attribute association. The listed classes, methods and attributes are only a small representative fraction of the whole software architecture.

4.1.2 RAII Wrappers

The OpenGL API requires a user to create and delete OpenGL objects, which often comes with allocations and deallocations of abstracted GPU resources. Missing destruction calls and thus missing deallocations are the cause of memory leaks. Combined with the incorrect use of the state machine bind operations, they can be a frequent source of software bugs. To mitigate this problem, we introduce a C++ class wrapper around each required type of OpenGL object. For this purpose, we use the *Resource Acquisition Is Initialization* or *RAII* paradigm [4]. RAII enforces the automatic deallocation of a resource by implicitly calling a destructor when the variable scope ends. Thus, in the wrappers constructor, we create the object with an `glGen*` call and have a `glDelete*` call in the wrappers destructor. We further encapsulate common operations concerning the resource into class methods. Each method binds its OpenGL object itself, which prevents confusions on what object should be bound at which point of time. This

is similar to the official RAII layer for Vulkan [5]. We show parts of such encapsulation in code 1 for OpenGL Buffer Objects.

```
template<typename T>
class GLBuffer {
  protected:
  std::size_t m_size = 0;
  GLuint m_handle = 0;
  GLenum m_target = 0;
  public:
  GLBuffer(const GLBuffer&) = delete;
  GLBuffer& operator=(const GLBuffer&) = delete;
  GLBuffer(GLenum target, std::size_t size)
    : m_size(size)
    , m_target(target)
  {
    glGenBuffers(1, &m_handle);
    bind();
    glBufferData(m_target, sizeof(T) * m_size, NULL, GL_DYNAMIC_DRAW);
    unbind();
  }
  GLBuffer(GLBuffer&& other) noexcept {
    m_handle = std::exchange(other.m_handle, 0);
    m_size = std::exchange(other.m_size, 0);
    m_target = std::exchange(other.m_target, 0);
  }
  GLBuffer& operator=(GLBuffer&& other) noexcept {
    std::swap(m_handle, other.m_handle);
    std::swap(m_size, other.m_size);
    std::swap(m_target, other.m_target);
    return *this;
  }
  ~GLBuffer() { glDeleteBuffers(1, &m_handle); }
  void bind() { glBindBuffer(m_target, m_handle); }
  void unbind() { glBindBuffer(m_target, 0); }
  ...
};
```

Code 1: RAII wrapper for OpenGL buffers. The OpenGL buffer only exists during the lifetime of the GLBuffer object and is moveable only. Methods bind and unbind the OpenGL object themself when needed.

A similar encapsulation is also introduced to OpenXR objects.

4.1.3 Hot Shader Reloading

As OpenGL shaders are loaded and compiled at run time, it is possible to create a system that enables the editing of shaders while the application is running. This way, immediately after saving the edited shader code file to the disk, the programmer can see the resulting changes

in the renderer. Without such a system, the programmer would need to restart the whole application to see the changes in action. As the application usually loads in data, e.g. 3D assets, from the disk, this can take multiple seconds or minutes. Additionally, required program state information, such as camera position, user interface settings etc. might get lost with restarts. To implement such a system, we create a watchdog thread that checks for shader file changes. When a file change is detected, the watchdog thread flags the OpenGL program as outdated in the main application thread. When the main application thread wants to use a program, it checks for the flag. In case the program is outdated, the main application sets the flag to updated, reloads the shader files from the disk and compiles them. If the compilation is successful, the application starts using the updated version of the shader. If the compilation fails, e.g. because of a syntax error, the application prints the error and continues with the old version of the shader.

4.2 Panorama Splitting

In order for our pipeline to process 360° panoramas as input images, we must split them into individual rectangular images that follow a pinhole model. Most commonly, such panoramas follow an equirectangular projection. A pixel in such panorama, given by the column index x and the row index y, represents a point on a sphere, described with two angles: The longitude χ and the latitude ψ. Given the panorama width w and height h, they are converted with:

$$\chi = \frac{2\pi x}{w}, \qquad \psi = \frac{\pi y}{h}. \tag{1}$$

The direction n, where the light that formed the pixel came from, can be computed with:

$$\vec{n} = \begin{bmatrix} \cos(\chi)\sin(\psi) & \sin(\chi)\sin(\psi) & \cos(\psi) \end{bmatrix}. \tag{2}$$

With the inverse mapping:

$$\chi = \operatorname{atan2}(n_y, n_x), \quad \psi = \operatorname{acos}(n_z). \tag{3}$$

The characteristics of a pinhole camera model are described by the ratio of between the image resolution and the focal length f. The longer f, the smaller the field of view becomes. Given the desired horizontal field of view angle of θ and the image width w in pixels, f in pixels is

given by:

$$f = \frac{w}{2 \cdot \tan\left(\frac{\theta}{2}\right)}. \tag{4}$$

A straight forward way for splitting a panorama is to create a *cubemap*. The six sides of a cube represent the views for up, down, forward, backward, left, and right. For no information overlap between the images, each cubemap view has a horizontal and vertical field of view of 90°. In section 2.4.2 we describe that we require the images to slightly overlap and suggest a field of view of a 100° to 110°.

To fill a whole split target image, we use backward mapping [6], thus loop over the target pinhole pixels and compute what pixel position to sample from the source panorama image to fill the target pixel. Therefore, we build $\vec{n}$ for each target pixel using the focal length as the z component, rotate it to the desired orientation of the image, compute the longitude and latitude of the rotated vector using eq. (3), and convert the angles to the panorama image coordinate by inverting eq. (1). Finally, we interpolate the corresponding pixels from the panorama image and write the interpolated pixel into the target image. We use bilinear interpolation for this step. Our tool also allows for splits with more than six images, which works analogously but will always come with some overlap in the target images. Figure 21 shows a panorama split into the six cubemap images.

(a) panorama (b) cubemap

Figure 21: Panorama Splitting. A 360° panorama (a) is split into 6 cubemap images (b) without any overlap.

4.3 Bibliography

[1] GLFW Community. GLFW. https://www.glfw.org/. Accessed: 2022-11-10.

[2] Omar Cornut. Dear ImGui. https://github.com/ocornut/imgui. Accessed: 2022-11-10.

[3] Khronos Group. OpenXR. https://www.khronos.org/openxr/. Accessed: 2022-11-10.

[4] cppreference community. RAII. https://en.cppreference.com/w/cpp/language/raii. Accessed: 2022-10-31.

[5] Khronos Group. vulkan_raii.hpp: a programming guide. https://github.com/KhronosGroup/Vulkan-Hpp/blob/master/vk_raii_ProgrammingGuide.md. Accessed: 2022-11-10.

[6] Thomas B Moeslund. *Introduction to video and image processing: Building real systems and applications*, pages 146–147. Springer Science & Business Media, 2012.

5 Conclusion

The goal of this thesis was to develop a GPU accelerated rendering application for displaying given IBMPP datasets and prototype novel techniques to support and further the scientific work of the project. As a result, we presented a very efficient image-based rendering application that builds upon traditional techniques. The extensions include a method for using multiple morphing cells to fill the viewport with information as much as possible and, to some extend, approximate parallaxes. Additionally, we explored a novel technique to handle color inconsistencies within the images of a dataset in real time. This color mapping technique imitates an eye slowly adjusting to new brightness levels, similar to visual effects in polygon-based game engines. While the dataset creation process is still subject of further research by different members of IBMPP, the rendering part of the technique is finalized and ready to be published. Additionally, we exploited the rasterization capabilities of graphics hardware to optimize the alignment quality of morphing meshes. We showed that the technique can reduce ghosting artifacts and sharpen the morphed result. As the processing of a whole scene only takes a few minutes, the technique will be part of the IBMPP dataset creation pipeline.

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Kuth Vorname: Bastian Matrikel-Nr.: 3609842

Fakultät: Informatik Studiengang: Informatik

Semester: Wintersemester 22/23

Titel der Abschlussarbeit:
Real-time Image-Based 3D-VR-Rendering of Open Environments

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☐ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,

☒ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: **https://www.th-nuernberg.de/datenschutz/**