Institute for Computer Graphics and Vision
Graz University of Technology
Graz

GPU-Accelerated Panoramic Mapping and Tracking

Master's Thesis

Georg Reinisch, BSc.
georg.reinisch@student.tugraz.at

December 2012

Supervision:

DI Dr.techn. Clemens Arth
Univ. Prof. DI Dr. techn. Dieter Schmalstieg

Abstract

Creating panoramic images in real-time is an expensive operation for mobile devices. Depending on the size of the camera image and the panoramic image the pixel-mapping is one of the most time consuming parts. This part is the main focus of this thesis and will be discussed in detail. To speed things up and to allow the handling of larger images the pixel-mapping process is transferred from the Central Processing Unit (CPU) to the Graphics Processing Unit (GPU). The independence of pixels being projected into the panoramic image allows OpenGL ES shaders to do the mapping very efficiently. Different approaches of the pixel-mapping process are demonstrated and confronted with an existing solution. The application is implemented for Android phones and works in real-time on current generation devices.

Keywords: Augmented Reality, Open Scene Graph, OpenGL ES 2.0, pixel mapping

Zusammenfassung

Das Erzeugen von Panoramabildern auf mobilen Geräten in Echtzeit ist eine rechenintensive Operation. Abhänging von der Größe des Kamera- und des Panoramabilds stellt das Pixelmapping den zeitaufwendigsten Teil dar, auf welchen im Zuge dieser Arbeit als Hauptfocus näher eingegangen wird. Um die Geschwindigkeit des Mapping-Vorganges zu erhöhen und um größere Panoramabilder zu ermöglichen, wurde der Mapping-Prozess von der Central Processing Unit (CPU) auf die Graphics Processing Unit (GPU) verlagert. Die Unabhängigkeit der zu projezierenden Pixel begünstigt die Verwendung von OpenGL-Shadern und ermöglicht einen effizienten Mapping-Vorgang. Die Arbeit befasst sich mit verschiedenen Pixelmapping-Methoden, welche einer bestehenden Methode gegenübergestellt werden. Die Anwendung wurde für Android-Handys entwickelt und läuft in Echtzeit auf derzeit gebräuchlichen Geräten.

Keywords: Augmented Reality, Open Scene Graph, OpenGL ES 2.0, pixel mapping

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Acknowledgements

I would like to thank my supervisor Clemens Arth for supporting me all the way during writing my thesis and providing a workplace for me at the Institute for Computer Graphics and Computer Vision (ICG) for the whole time. Therefore also thanks to my room colleagues Stefanie Zollmann and Jonathan Ventura for letting me stay. In general I want to thank the ICG at the Graz University of Technology, especially Markus Tatzgern and Lukas Gruber for their support in OpenGL ES and OpenSceneGraph.

Special thanks go to my parents that supported me during my whole period of study and without them all this would not have been possible.

Thanks to my girlfriend and my friends, who always motivated me and listened to all the problems I had during the development.

Contents

Chapter 1

Introduction

Mixed Reality (MR) ranges from *Virtual Environment* (VE) to *Real Environment* (RE). Whereas VE, or more commonly called *Virtual Reality* (VR), is a completely virtual continuum, RE contains only real objects from a real world scene, e.g., a video taken by a camera [Milgram94]. In between those two extrema, *Augmented Reality* (AR) describes a state of a real environment with virtual objects superimposed or merged with it. AR systems do not only have to combine virtuality and reality, but have to be interactive and registered in 3D. The motivation for creating AR-systems is that they can enhance the human perception through interaction with the real world [Azuma97].

The fast improving technology on mobile/smart-phones including better cameras, GPS sensors, gyroscope sensors, faster central processing units with multiple cores and even graphics processors enables new possibilities for AR. Despite all this innovations, using AR systems outdoors still raises a number of difficulties, because there is less control over the environment, compared to indoor environments [Wagner10]. Visual tracking methods (e.g., feature point extraction, optical flow) do not work well if homogenous areas like a gray sky or big, close walls occupy a large amount of the camera view. Since rotations are the main source of inaccuracy in the tracking process, integrating a gyroscope sensor can improve the robustness of trackers considerably [Schall09].

Real-time tracking in 6DOF on mobile phones for AR purposes stays a difficult and computationally demanding task for the time being. Tracking in 3DOF (ie., tracking of rotation only) using panoramic images was demonstrated to work very well for AR recently [Wagner10].

For creating panoramic images several different kinds of techniques are proposed in the literature. Hardware intensive approaches for capturing panoramic images are realized with a lens that has a large field of view, such as the fisheye lens. Other approaches use parabolic mirrors or mirrored pyramids. Creating panoramic images out of several regular photographs requires the images to be composed and aligned using

stitching or image mosaic algorithms [Szeliski97]. Most of the recent approaches that generate panoramic images do this in an offline process [Steedly05, Szeliski97]. After taking pictures with a camera, the images are read in by an application that composes them into one single panoramic image. A good overview of several image alignment and stitching algorithms is given in [Szeliski06].

For AR purposes, Wagner el. al. created a method that captures an image with the camera of a mobile phone and maps it onto the panoramic image in real-time [Wagner10]. The approach takes the camera live preview feed as input and continuously extends the panoramic image, while the rotational parameters of the camera motion are estimated. If the panoramic image is filled, it can be saved and no further post-processing step is required.

While the process of mapping pixels is done by the CPU in the approach of Wagner et. al., this work is dedicated to the development of a mapping process that runs entirely on the mobile phone GPU. General purpose programming on the GPU is a common practice in the field of image processing, since processing a pixel's color value can often be heavily parallelized. Furthermore, post-processing steps for enhancing the quality of the panoramic images can be performed automatically and directly on the dedicated graphics processing hardware.

1.1 Problem Statement

In order to achieve a satisfying result in mapping and tracking a panoramic image in real-time, a compromise between render speed and image quality has to be made. This is especially true when computing the image on a handheld device with very limited resources. Even the strongly increasing computational powers of mobile phones cannot fully remove the limits if the amount of image data becomes too large.

The existing panoramic mapper and tracker by Wagner et. al. solely works on the CPU. Since mapping a camera image onto a panoramic map is one of the most expensive operations computed on a CPU, only a small image resolution can be mapped. Increasing the size of the camera image has a significant and adverse impact on the render speed of the mapping process. Therefore, the application is accelerated by only mapping new pixels, to keep the number of pixels to map as low as possible. A downside of this approach is that it eliminates the chance of blending pixels to cover seams generated by brightness differences.

Additional functionality, like clearing an area of the current state of the panoramic image, is a costly operation which has to be performed on the CPU. This feature is of special interest, because it allows the removal of a person or a moving object covering a part of the panoramic scene. Without this feature a new attempt of taking an image has to be made, which can be very frustrating and negatively impacts the usability of the approach.

1.2 Motivation

Due to developments in the last few years, nowadays mobile phones or 'smart-phones' have faster processors with multiple cores and are equipped with GPUs. Furthermore the OpenGL ES 2.X standard allows developers to customize the programmable graphics pipeline using shader programming [Munshi10]. These improvements enable new possibilities of mapping the camera image onto a panoramic map on mobile phone GPUs.

Using shader implementations for general purpose computation is very useful for operations that can be heavily parallelized on GPUs. Especially for image processing, pixelwise calculations are often computed in a fragment shader. If they behave linearly, the calculations can be done in the vertex shader, where they are automatically interpolated between these vertices and accelerated by the devices' hardware. Both shader run as separate programs for each vertex (vertex shader) or fragment (fragment shader). Transferring the calculations to the GPU reduces the computational expenditures of the CPU and partitions computation between the CPU and GPU. Despite GPUs on mobile phones having very limited computational power compared to the ones integrated in PCs, generating free resources on the CPU enables to extend functionality without loosing rendering speed.

Having all these hardware and functionality improvements, the goal of this work is to complement the CPU-based rendering approach by Wagner et. al. with a GPU-based implementation and to transfer computational costs from the CPU to the GPU. By using shaders, one is enabled to calculate the image mapping in parallel. Overlapping mapping areas can be blended and therefore the visibility of seams and artifacts can be reduced or eliminated. Furthermore operating with shaders facilitates the removal of unwanted objects in the scene.

1.3 Contribution

In this work we demonstrate a system that tracks the current camera orientation on the CPU and maps the camera image according to its orientation using the GPU in real-time.

The tracking part is based on an approach developed by Wagner et. al. [Wagner10]. They introduce a system that realizes the tracking of a camera image and the creation of a panoramic map in real-time solely on a mobile phone's CPU. As discussed in Chapter 3, the rotational movement of the camera is estimated and used for the mapping process. In contrast to this pure CPU-sided approach the camera orientation is passed to a shader implementation (in form of a rotation matrix), which is described in Chapter 4 in detail.

The advantages of this GPU-mapping approach are on the one hand the parallel processing of pixels and on the other hand the efficient way of improving the image quality. In Section 5.2 image refinement methods are discussed that reduce or eliminate seams and artifacts generated by mapping camera images of different brightnesses.

The approaches for improving the image quality are tested with regard to the general impression of the outcome, the tracking quality of the newly generated panoramic image and the render speed. The results are interpreted and compared with results of the CPU-mapping. In terms of speed significantly larger panoramic image sizes are tested.

To enhance the user-friendliness of taking panoramic images, a wiping function is added. This function allows the user to remove unwanted areas of the panoramic image and remap them again. This can be very useful, if a moving object or person covers an important part of the scene.

1.4 Architecture of the Studierstube ES AR framework

Because the described work builds upon existing software, the following paragraphs are dedicated to an overview about the StbES framework, developed by the handheld AR team [ICG09] of Graz University of Technology.

The predecessor of the AR framework was Studierstube 4, which was not suitable for mobile phones and therefore was rewritten and optimized to achieve real-time

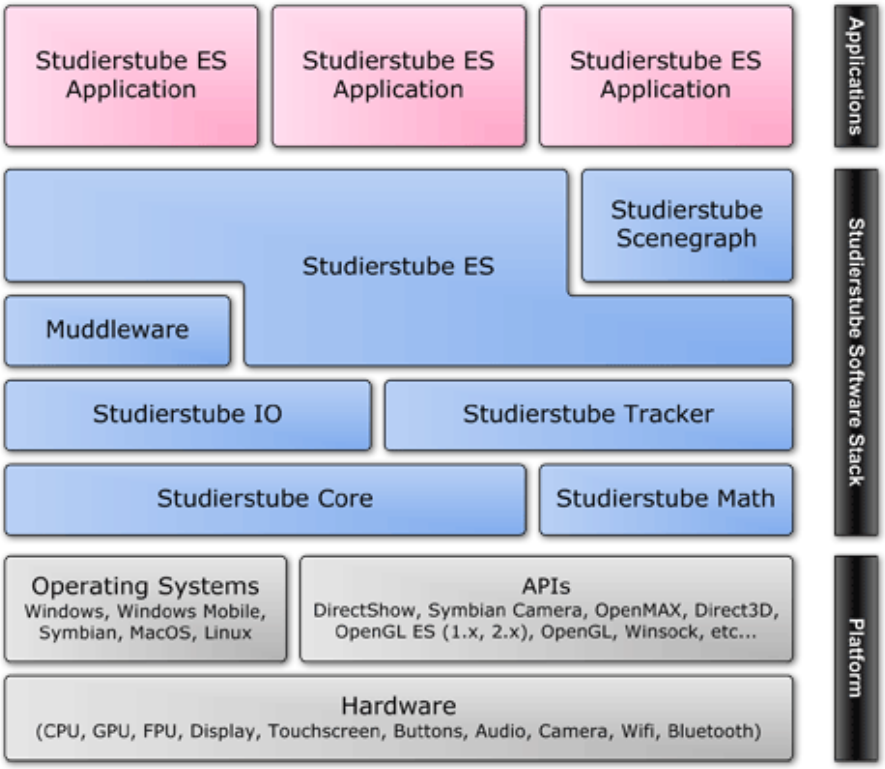


Figure 1.1: Architecture of the Studierstube ES AR framework

frame rates for AR-applications on mobile phones. A detailed description can be found in [Wagner09a, Wagner09b].

As seen in Figure 1.1 the framework supports several platforms. It abstracts different hardware types of mobile phones, operating systems and special APIs such as Direct 3D, OpenGL ES, etc. The framework is divided into modules, for which each of it provides special functionality to the programmer.

The math module can be used for mobile phones that do not have an integrated Floating Point Unit (FPU). It enables the calculation with fixed point arithmetic, which can speed up applications on such devices. For PC usage the arithmetic can be switched between fix-point and floating-point. The core module takes care of the thread handling, sockets, logging and handles diverse sensors. It also supports general data types like vectors and strings to be platform independent. In the IO-module HTTP-requests and zipping/unzipping of files is handled as well as the implementation of XML- and string-parsing functionality. The tracker module supports tracking of different markers and natural feature tracking and is responsible for camera pose estimation. For efficient rendering of a scene the scence graph module offers functionality for creating a scene graph. To put all modules together and to create an

entry point in an application the framework's Studierstube ES module is used. For creating an application the initialization, update and render methods, which have to be inherited from the *Application* class can be overwritten and customized for one's own needs.

In this work however, a different render module is used, since the scene graph module is very likely to be replaced by the one of the most popular open source cross platform tool kits, OpenSceneGraph (OSG).

1.5 OpenSceneGraph

OpenSceneGraph is an open source render tool kit that works on several platforms and is used for developing high-performance graphics applications. Similar to the Studierstube SG module, the OSG is based around the concept of a scene graph. An object-oriented framework is provided on top of the OpenGL functionality [OSG07].

The architecture of OSG described in [Wang10] is designed to be highly scalable to enable runtime access to extended functionality. The core component of OSG shown in Figure 1.2 consists of four libraries. The first one is the OpenThreads-library to provide a minimal and complete thread interface. The second library that provides all the basic elements for building a scene graph, is the osg-library. In the osgDB-library a plug-in mechanism is responsible for reading and writing from or to files and stream IO-operations. The osgUtil-library is designed to build the OSG rendering backend. It traverses the scene graph and performs culling and converts the scene into OpenGL calls.

In addition to the core component several modular libraries, called NodeKits, support the framework. These libraries include, for instance, a support to render animations, special effects, particle-based effects, geographical terrains, text, etc.

In most cases the libraries from the core component are the only libraries required for building an OSG-based application.

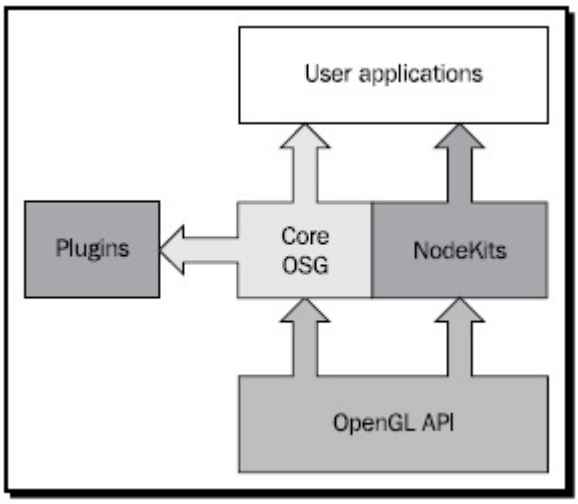


Figure 1.2: Architecture of the OpenSceneGraph tool kit by [Wang10]

Chapter 2

Related Work

2.1 Panoramic Imaging and Image Mosaics

The generation of panoramic images is a popular topic in the literature. For image alignment several approaches exist that are suitable for different types of cases. A tracking method described by [Lowe04] that searches for scale invariant key points (SIFT), is used in several offline approaches. To generate such features, SIFT uses a scale-space extrema detection to search for features over all scales and image locations. At the candidate's location key points are selected to measure the stability in location and scale. In an orientation alignment step orientations are assigned to the key point locations based on local image gradient directions. A descriptor for each key point is computed by measuring the local image gradients at the selected scale in the area around the key point. Most existing approaches for panoramic imaging or creating image mosaics work offline [Brown03, Steedly05, Szeliski97].

Adam et. al. in [Adams08] discuss a method in which the successive images of a camera's view finder are aligned online and in real-time. Therefore, by aligning integral projections of edges in two images, the inter-frame translation is estimated. For refining this estimation the point features are aligned and a full 2D similarity transformation is computed. However it does not permit to create closed 360 degree images and tracking the 3D motion of the phone. In [Xiong09a] the viewfinder algorithm is used for tracking the camera motion to create high resolution panoramic images. Every time the camera motion exceeds a threshold with respect to the previous image a high resolution image is captured. The whole approach itself does not run in real-time and requires offline processing.

A big problem of several approaches for panoramic image creation is that important parts of the scene are missing in the final image since they have been cut off. This

problem arises if the user misses to capture relevant areas or the relevant areas disappear when cropping the panoramic image to its rectangular shape. The panoramic image preview described by Baudisch et. al. in [Baudisch05] shows a low-resolution real-time preview while shooting the image that is similar to the approach in this paper.

A similar real-time tracking method as in [Wagner10] is described by DiVerdi et. al. in [DiVerdi08]. The system called *Envisor* is capable of creating environment maps online. For refining optical flow measurements DiVerdi et. al. use landmark tracking. The result of the tracking process is frame-wise projected into a cube map. To avoid having gaps in the final result the user gets a feedback of the current state of the panoramic image and the remaining gaps are filled by texture diffusion when finishing the image. Since landmark tracking is a costly operation, it does not run on mobile phones, but requires extensive GPU resources for real-time processing.

The real-time panoramic mapping and tracking approach by Wagner et. al. combines the tracking of the camera orienation and panoramic image creation and is still able to run in real-time. Therefore it can be distinguished from the approaches described above.

Approaches that do not mainly focus on generating panoramic images, but generate them in form of a by-product are based on Simultaneous Localization and Mapping (SLAM) [Davison03]. An approach that is based on SLAM is a visual compass created by Montiel et. al. [Montiel06], which is able create a sparse 3D reconstruction of the environment. Another SLAM-based approach for augmented reality that even runs on mobile phones was developed by Klein and Murray [Klein07]. Their approach supports 6DOF and can handle a few hundreds of feature-points, whereas the tracker described in [Wagner10] has only 3DOF but can handle thousands of feature-points.

2.2 GPU-acceleration

For the mapping part GPU-accelerated approaches exist that already use the support of the graphics processor for image blending. López et. al. developed a *document panorama builder*, which takes several low resolution viewfinder images from a video of a document for interactively creating an image mosaic that reduces blurry artifacts [López09]. While the quality evaluation and frame selection runs online, the stitching and blending is done after finishing the video and is accelerated using the

GPU. Since López et. al. used OpenGL ES 1.1 they are not able to have the flexibility of programmable shader, but still point out to gain speed for parallelizing the processes.

Pulli et. al. warp their images via spherical mapping calculated on the GPU [Pulli10]. They use OpenGL ES 1.1 to reduce the computational time in order to run the application on smart-phones. The input images are divided into a triangular mesh and the geometrical transformation is calculated for each vertex. The original image is provided as a texture map to the GPU and after warping the result, it is read back to the CPU for further calculations.

An approach that creates spherical image mosaics in real-time using graphics processors for faster computation is discussed by Lovegrove et. al. [Lovegrove10]. They realize an efficient second-order method for parallel image alignment and a global optimization for a map of key frames over the whole viewsphere. Both the CPU and GPU are used for calculations. For rendering the scene on the GPU, the color values of each pixel are summed up. Then the current value is divided by the alpha value, which is used as a mapping counter. However this approach does not run on mobile phones in real-time, since the computing power of handheld devices is very limited.

2.3 Image Refinement

Removing the seams of panoramic images that occur if two images with different illumination are stitched together is a widely discussed topic.

Uyttendaele et. al. calculate regions of differences (ROD) to locate dynamic areas of the scene, which cause ghosting artifacts. By eliminating certain RODs the ghosting artifacts can be removed. To cope with the differences in brightness a transfer function is applied to the input image to smooth transitions between its neighbors. If the amount of neighboring images increases, the functions would get too complex. Therefore a block-based exposure adjustment technique is used to vary the image's transfer function [Uyttendaele01].

Tian et. al. introduce a method for color correction that is based on a histogram map. The method acquires the color histogram of one of the images that share an overlapping area. An estimated color transform matrix is generated and applied to the other image of the overlapping area [Tian02].

In the approach of Xiong et. al. graph cut is used for optimal seam finding in the

overlapping areas of neighboring images to reduces the appearance of brightness differences. If this method is not able to remove the seam completely additional gradient domain transition smoothing is applied globally. The approach is divided in a sequential image blending process for mobile devices and a global image blending process for a better solution on a global scale [Xiong09b].

Xiong et. al. in [Xiong10] discuss a fast stitching approach for composing several source images into a panoramic image. This approach has little memory consumption and is able to run on mobile phones. The color correction for balancing colors and luminance in the whole image sequence is achieved through dynamic programming. With this method optimal seams between adjacent images are found in the overlapping area and merged together. For further smoothing color transitions image blending is used.

Despite all the offline approaches, no image refinement approach has been found that completely removes seams and ghosting artifacts and runs in real-time, especially not on mobile phones. Additionally most of the approaches use all captured images for refining the panoramic outcome, which requires a lot of memory and is hard to realize for achieving real-time frame rates.

The real-time approach described by Lovegrove et. al. [Lovegrove10] sums up the pixels' color values and divides them through the number of times the pixel has been mapped. This approach requires an exact tracking algorithm, otherwise edges and structures appear blurry. Furthermore seams of strong differences in brightness will be reduced, but are not eliminated completely.

Pulli et. al. in [Pulli10] use a fast image cloning approach for transition smoothing based on [Farbman09]. Farbman et. al. introduce an alternative coordinate-based approach to solve a Poisson equation with Dirichlet boundary conditions. The Poisson equation has to be solved for a large linear system that interpolates the differences between the boundary of the source image and target image across the cloned area. In the approach of Farbman et. al. the value of the interpolant of this interpolation is given by a weighted combination of values along the boundary for each interior pixel. This mean-value coordinates based image cloning approach runs in real-time for desktop-GPUs and delivers seamless results, but cannot be computed online on mobile phones.

[Degendorfer10] implemented a brightness correction method to enhance the image quality with an extended dynamic range. Therefore they calculate the lighting for the corresponding feature points and add the average difference to the color values of the panoramic image (16-bit color channel). Since for saving the image only eight

bits per color channel are available, the colors have to be adapted and the image appears more gray. The strength of the seams is reduced, but they are not eliminated completely.

2.4 Demand for Action

All approaches mentioned in this chapter are either not running in real-time on mobile phones or cannot eliminate artifacts completely, such as ghosting or brightness seams. The approach by Wagner et. al. realizes real-time panoramic mapping and tracking, but does not address removing brightness artifacts. The method described in [Pulli10] achieves seamless mapping of adjacent images and removes ghosting artifacts, but is not able to run in real-time.

Since mobile devices have limited resources, the blending approaches discussed in this paper cannot remove all seams and artifacts completely either, but deliver a significant improvement of the image quality and mapping and tracking still runs in real-time on mobile phones.

Chapter 3

CPU-Side Panoramic Mapping and Tracking

The starting point for this paper is the already existing mapping and tracking approach by Wagner et. al. [Wagner10] that is mentioned in the previous chapter. In this chapter this approach is described in more detail to understand how the CPU-side tracking and mapping procedure is realized. The main advantage of this tracker is that it combines the panoramic mapping and orientation tracking on the same data set on mobile phones in real-time. The approach runs at 30Hz on current mobile phones and is used for various applications, such as the creation of panoramic images, offline browsing of panoramas, visual enhancements through environment mapping and outdoor Augmented Reality.

3.1 Panoramic Tracking

To estimate the location of the current camera image for the mapping process, the new image has to be tracked accurately. Therefore Wagner et. al. [Wagner10] use the FAST corner detection [Rosten06] for feature point extraction. The found feature points are ranked by strength. To get a valid tracking result the amount of the corner points must exceed a given threshold. The strongest corner points (key points) are kept for further tracking. This procedure is done for a low, medium and high resolution level (see Figure 3.1). If the threshold is not achieved at the medium or high resolution level, the tracker has to be re-initialized.

For tracking the key points a motion model is used, which estimates the new orientation of the camera in a new frame. The difference in orientation between the currently mapped camera image and the previous one is used to calculate the direction and velocity of the camera. Using the estimated orientation the current frame extents are projected back onto the map and the key points in the area are extracted. Backwards-mapping them into the camera image eliminates the key points that are projected outside of it. As a support area of a feature point, 8x8 pixel patches are used and are warped back such that they correspond to the camera image. Since template

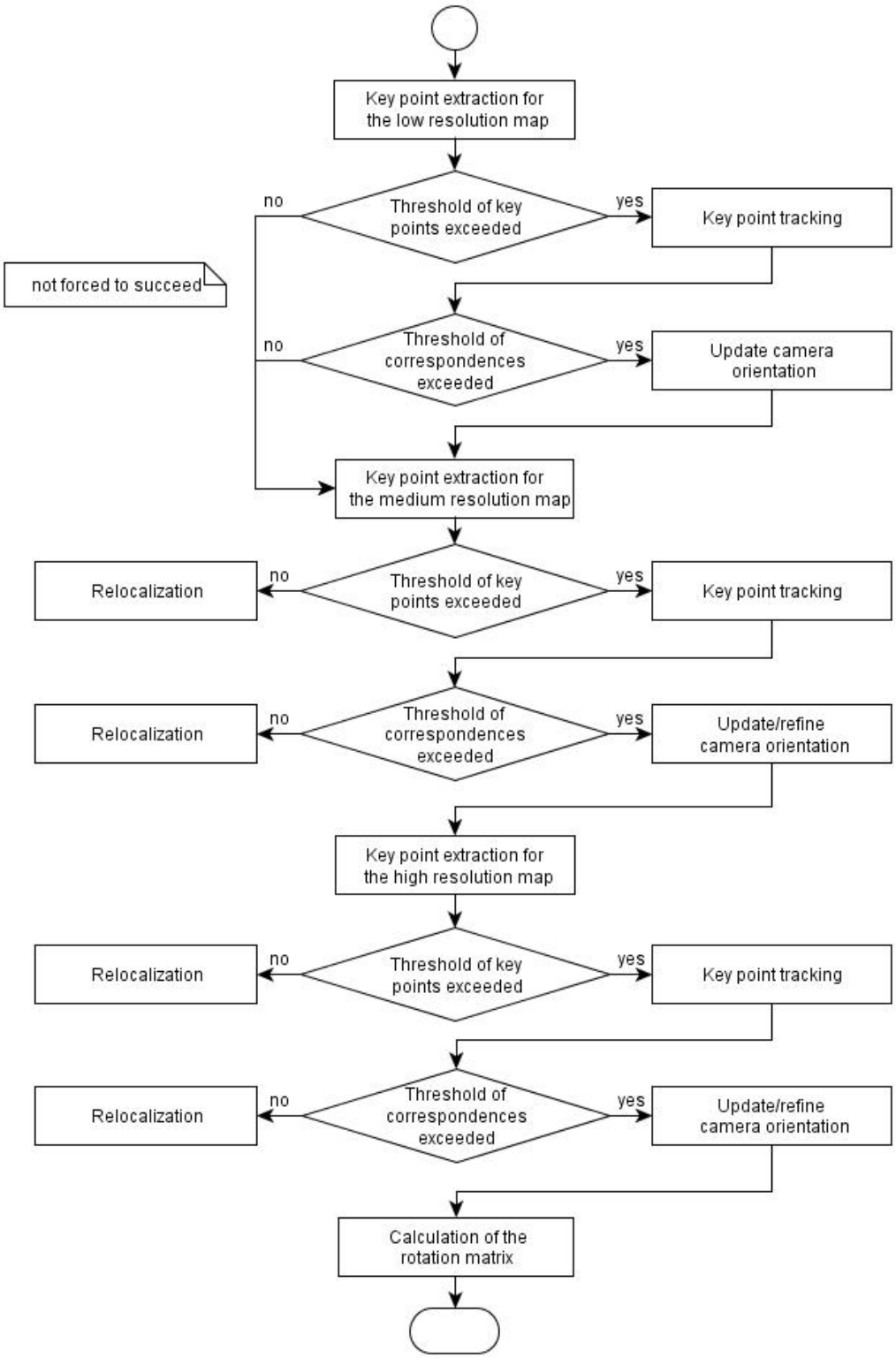


Figure 3.1: Tracking process for each render cycle

matching is a computational costly operation, the matching is done in lower resolution images first. The orientation is then refined by matching the key point from the lower resolution image with the next higher one and finally with the original size. If not enough correspondences are found the tracker fails to update the orientation and a re-initialization step is invoked. If the tracking succeeds, the output generated by this process is a rotation matrix that is further used in the mapping process to project the current camera image onto the map.

The tracking process searches for matching key points only between the current camera frame the yet mapped panoramic map. A pure tracker, however, is not able to re-initialize from an arbitrary orientation. Since the tracker can fail to find the new orientation for a new frame due to a lack of found or matching key points, a relocalization procedure is required. The tracker stores low resolution key frames and their corresponding camera orientation. This background operation runs during the creation of the map. Once the tracker is lost, normalized cross correlation is used to compare the current camera image to the key frames. For a more robust localization of the camera image, both the frame and the image are blurred. This is similar to the approach of using small blurry images, as it is used in parallel tracking and mapping (PTAM) for example [Klein07].

3.1.1 Brightness Offset

Degendorfer [Degendorfer10] calculates a brightness offset of the current camera image to the panoramic map in his master's thesis to compensate for abrupt brightness changes due to an unexpected change of exposure in the camera image. While comparing the feature points for correspondences, the brightness values are calculated from the respective color values at these points. These brightness values are used to reduce the differences in brightness between the camera images and the panoramic map by adapting the color values of the newly mapped pixels with this offset.

This approach is part of the image refinement discussed in Section 5.2.1.

3.2 Panoramic Mapping

Mapping an environment can be done by several types of maps. Cube maps cover the whole environment but have discontinuities at the edges. To solve that problem one could use spherical maps at the cost of high non-linearity. As an alternative

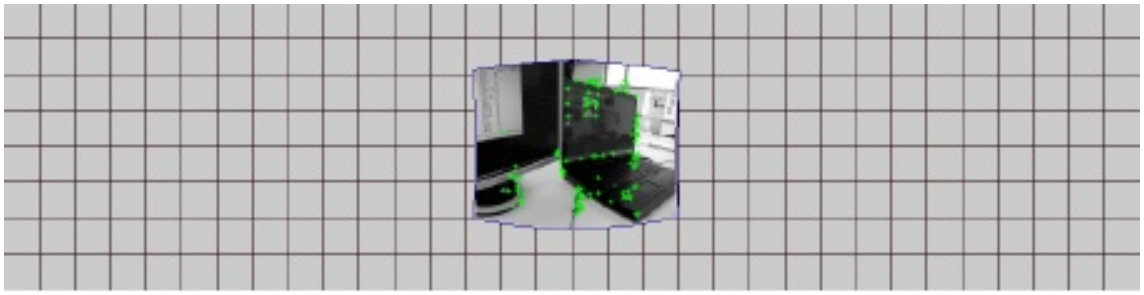


Figure 3.2: Map with grid after the first frame has been taken. The green dots represent the key points found in the frame. [Wagner10]

solution with noticeable computational savings a cylindrical map has been chosen. A cylinder can easily be mapped to a texture and has only one discontinuity on the left and right border of the panoramic image. The regions of the environment pointing to the top and bottom are neglected since they have usually no practical use in taking a panoramic image.

The panoramic map is split up into a regular grid of 32x8 cells (see Figure 3.2), which simplifies the handling of an unfinished map. During the mapping process the cells get filled with mapped pixels. As soon as a cell is completely mapped it is marked as completed, down-sampled to a lower resolution and key points are extracted for tracking purposes. To increase the robustness of the tracker, two lower resolution maps are created as well.

To project the camera image onto a map accurately, the intrinsic and extrinsic camera parameters have to be known. Since the camera used in this approach does not change the zoom or focus, the intrinsic parameters of the device can be calculated offline. Recent mobile phones are able to automatically correct radial distortions to a limited degree. To further adjusting these parameters a calibration toolbox can be used (e.g. the Caltech camera calibration toolbox [1]) that measures the parameters from pictures taken of a calibration pattern. Additionally artifacts like vignetting can be corrected as well. Those artifacts arise because camera sensors are dependent of the angle of the incoming light. The further the pixel is away from the image center the steeper is the incoming light, which results in a darker appearance of pixel. To measure this effect an image of a diffusely lit whiteboard can be taken.

For mapping the camera image onto the cylinder pure rotational movement is assumed and therefore 3DOF are left to estimate the correct projection of the camera

[1] `http://www.vision.caltech.edu/bouguetj/calib_doc/`

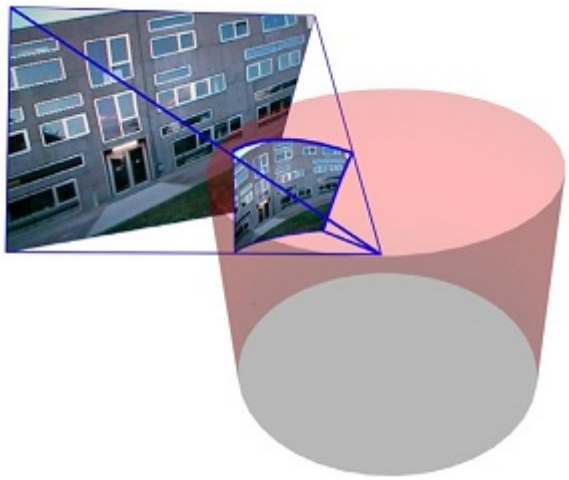


Figure 3.3: Projection of the camera image on the cylindric map. [Wagner10]

image. This might not always be true for practical use, but regarding the distance to the scene the translational movement can be neglected. Especially for a trained user parallax errors are minimized.

The mapping process starts after estimating the orientation of a camera image in the map. The rotation matrix calculated by the tracker is used to project the camera frame onto the map. The corner pixel coordinates of the camera image are forward-mapped into map space and the put up area by the frame represents the estimated location of the new camera image. Due to radial distortions during the mapping process the frame is not rectangular and cannot be accurately represented by only four corner coordinates. To achieve a smoother frame curve, additional corner points are added along the edges. Each edge between the four main corners is divided into three parts where the additional points are integrated in the frame to achieve higher accuracy.

Since forward mapping the pixels from the camera frame to the estimated location on the cylinder can cause artifacts, the camera pixel data has to be backwards-mapped. Even though the mapped camera frame represents an almost pixel-accurate mask, pixel holes or overdrawing of pixels can occur.

Mapping each pixel of this projection would generate a calculation overload since for a 320x240 pixel image more than 75,000 pixels have to be mapped. By reducing the mapping area to the newly mapped pixels (only those pixels where no image data is available as shown in Figure 3.4), the computational power is reduced significantly.

To check whether a pixel is in the projected camera frame and has already been

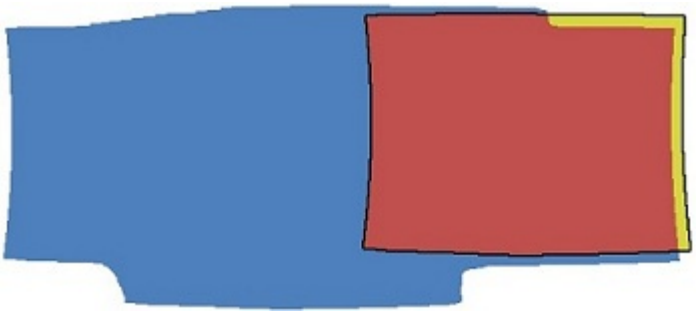


Figure 3.4: Mask created due to camera rotation marked with a black frame. Blue is the area that has already been mapped. Red marks the area that has already been mapped and falls to the mask. Pixels to be mapped are marked in yellow. [Wagner10]

mapped can solved by using a mask with one entry for each pixel. This method is sufficient to filter the pixels that need to be mapped, but results in a too slow and memory intensive process. To avoid this effect a run-length encoded (RLE) mask is used to store zero or more spans per row. Each span defines which pixels of the row still needed to be mapped and which is not by storing its left and right coordinates. Using Boolean operations to compare the left and right coordinates of two spans, the check if a pixel needs to be mapped or not can be applied very efficiently.

Chapter 4

GPU-Accelerated Mapping

Since the development of OpenGL ES 2.0 the programmer has more control of rendering a scene. Especially for general purpose GPU applications it is very useful to be able to access each vertex and fragment in a respective shader program. During the mapping process several parts can be easily parallelized and hence are ideal to calculate on the GPU. Since the forward-/backward-mapping described in the previous chapter is completely independent for each individual pixel, the idea is to compute the mapping part in a shader-program on the GPU. Furthermore pixel operations like blending pixels or clearing certain pixels can be accomplished with little effort using shaders.

4.1 Structure of the Mapping and Tracking process

Representing the structure in form of a scene graph shown in Figure 4.1 gives an overview of the implementation of the GPU-based mapping discussed in this section.

Root Node (brown): The Root node connects the CPU-based tracking/mapping and the GPU-based mapping with the preview nodes. The generated panoramic maps are acquired by this node and passed along for displaying. Therefore it is possible to switch between the CPU-based mapping result and GPU-based mapping result in the preview.

CPU-branch (green): The CPU-node represents the wrapper node for the functionality implemented for the CPU mapping and tracking. It contains the initially prepared data that is required for the tracker. To initialize the tracker the allowed window sizes and the calibration file data is read from a configuration file. In the camera calibration file intrinsic camera parameters are defined (focal length, principal points, distortion parameters) as well as the used camera image size.

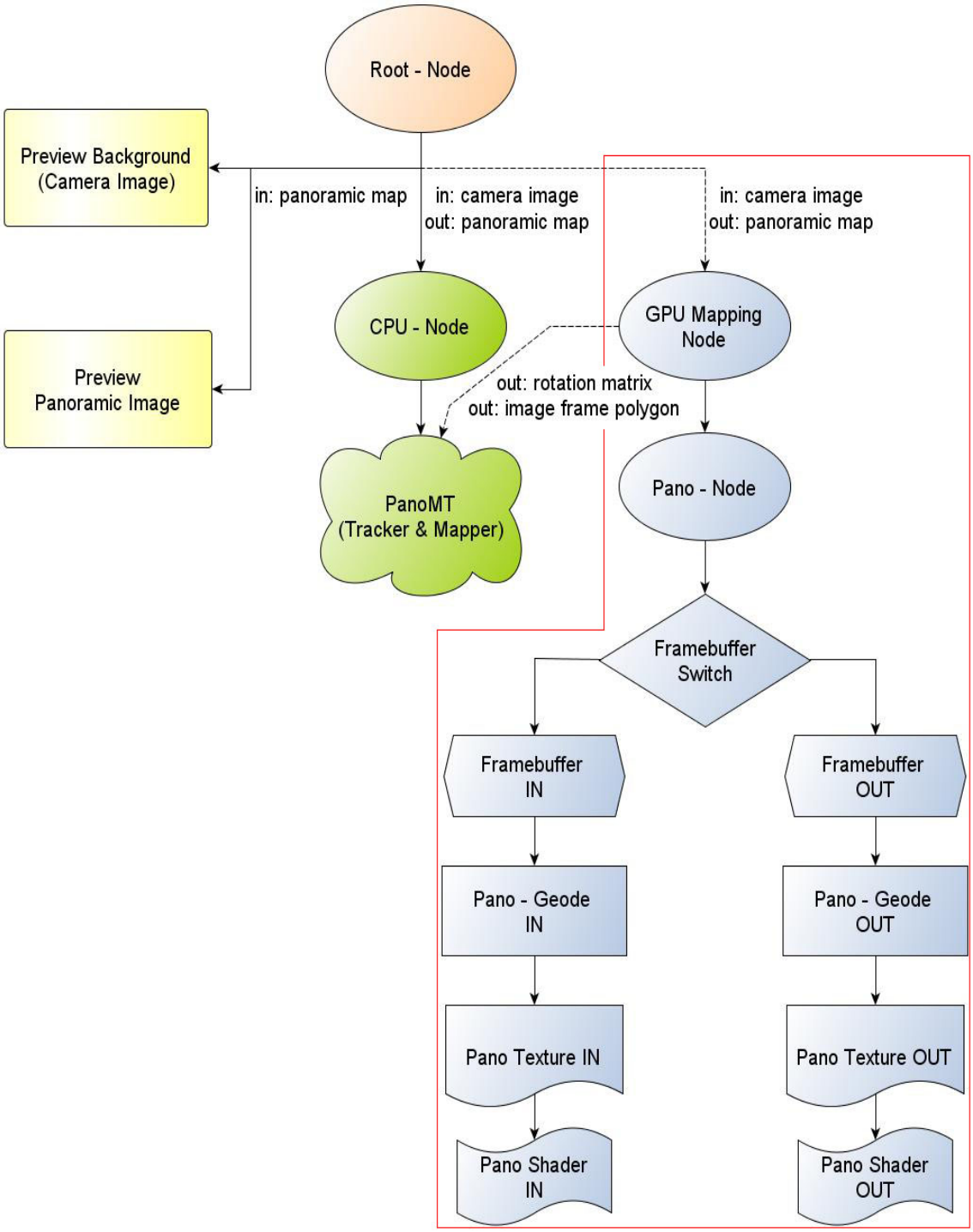


Figure 4.1: Structure of the GPU-Mapping in a scene graph view. Brown: root-node, green: CPU-based tracking and mapping, blue: GPU-based mapping, yellow: preview of the current camera image and the panoramic map

The main part of the CPU-based operations is the update traversal of the tracker. For every camera frame, given that the initialization process has finished successfully, the image is checked for resizing depending on the entry in the configuration file. If the input image is the same size as expected the scaling step is skipped. Since the tracking is realized with gray scale images to achieve higher frame rates, the image must also be converted into a gray scale image. After that the camera images (colored and gray) are prepared for the tracker update.

During the tracking process the rotation matrix and the estimated projection area of the camera image is calculated, which is used by the GPU-based mapping.

GPU-branch (blue): The blue part of Figure 4.1 represents the GPU-based mapping part, which is the main focus of this work. The GPU-mapping-node represents the wrapper node for the GPU-side mapping. It contains the initially prepared data that is required for mapping, like initializing the background image, the mobile phone's camera and the calibration of the camera. Furthermore the rotation matrix as well as the estimated projection area of the camera image, computed during the tracking process, are acquired and passed to the Pano-node.

The Pano-node organizes the switching of the framebuffers, where the actual mapping process takes place (see Section 4.2). It controls which branch of the switch is active and used for writing to and which is inactive and used for input information. Additionally the node passes the CPU-information fetched by the GPU-mapping-node along to the mapping process.

Scene Background and Panoramic Preview (yellow): In this branch the camera image of the live preview feed as well as the preview of the current state of the panoramic image is displayed. The displaying process is independent of the panoramic image generation, since the connection between the generation and displaying of the panoramic images is handled by the Root-node. The scene background displays the current camera image that represents the input for the creation of a panoramic image. Besides the background scene the progress of the panoramic image is displayed, which helps the user complete the image without gaps and gives information about the current location of the camera.

4.2 Framebuffer Switching

To be able to map only those pixels that have not been mapped before, information about the current progress of the panoramic image is required. Therefore a

render to texture approach using two framebuffers has been chosen. Continuously updating and displaying one texture in one render cycle using OpenGL ES 2.0 is not possible. It can either be read from a texture or written to it. Due to this circumstances the use of a common method also known as ”ping-pong technique” is required.

To realize this technique two framebuffers and two textures (one input texture and one output texture) are created. Each texture is assigned to a framebuffer. Whereas the output texture is used as render target, the input texture is used to determine which parts of the camera image are already mapped and which parts have to be updated. In addition information for image refinements can be retrieved from the input texture. After each render cycle the framebuffers with their respective textures are switched. The former output texture becomes the new input texture and vice versa.

The mapping itself is done in the shader OpenGL ES 2.X supports.

4.3 Shader Mapping

Using shaders allows the programmer to manipulate each pixel without being bound to the fixed function pipeline. As OpenGL ES 2.0 supports vertex and fragment shaders, these two are used to map the camera image onto the panoramic map. The vertex shader program is processed for every vertex of a geometry and the fragment shader program is processed for every fragment of it. A property of the GPU-mapping is that every mapped pixel can be handled separately, which makes shader programs ideal for this task. GPUs usually consist of several cores, where pixel data can be processed in parallel. This increases the speed of the mapping process considerably. One GPU-core by itself has a much lower computational capacity than the CPU.

The vertex shader is used to map the panoramic texture coordinates on the respective vertices of a plane. The texture coordinates between the vertices are interpolated and passed on to the fragment shader, where each fragment can be manipulated and written to its coordinate in the framebuffer.

In the fragment shader the color values for each fragment are determined. For the mapping part the required information consists of the current camera image deployed as a texture, the coordinates of the fragment the shader-program is processing, the panoramic image available as another texture and mathematical in-

formation of the camera orientation (i.e. the rotation matrix calculated by the tracker).

In general, every pixel of the panoramic image is mapped separately in its own shader program run. This means that for each pixel it is calculated if it lies in the area where the camera image is projected or not. If the pixel lies in this area, the color of the respective pixel of the camera image will be stored at this location. Otherwise the pixel of the input texture is copied to the output texture.

4.3.1 Mathematical description of the mapping process

To prepare the shader data, as many of the required calculations as possible are calculated before the information is passed to the fragment shader. It is crucial to keep the number of calculations in the shader to a minimum, since it will be executed for each fragment and will amount to huge computational costs in total. All the information that does not vary across the separate fragments is prepared outside the fragment shader. This information contains the panoramic image resolution, the camera image texture and camera image resolution, the rotation matrix, ray direction, the projection matrix and the angle resolution. Using this information the mapping calculations can be efficiently performed in the fragment shader.

To calculate the angle resolution, the model for the parametrization of the surface needs to be known. As suggested by Wagner et. al. [Wagner10] we chose a cylindrical model for the mapping procedure. A cylindrical projection can be mapped to a texture and has only one discontinuity on the left and right border. The radius r of the cylinder is set to 1 and the circumference C is therefore $2 \cdot \pi \cdot r$. The ratio of the horizontal and vertical size is chosen to be 4 by 1 and the height h of the cylinder is therefore set to $C/4.0$. The ratio of the panoramic map can be chosen arbitrary (e.g. 8 by 1, 4 by 3, etc.), however, it is recommended to use power-of-two values, since GPU textures prefer such texture sizes. The angle resolution for the x-coordinate a is composed by the circumference divided by panoramic texture width W and for the y-coordinate b it is composed by the cylinder height divided by the panoramic texture height H as follows:

$$a = \frac{C}{W} \tag{4.1}$$

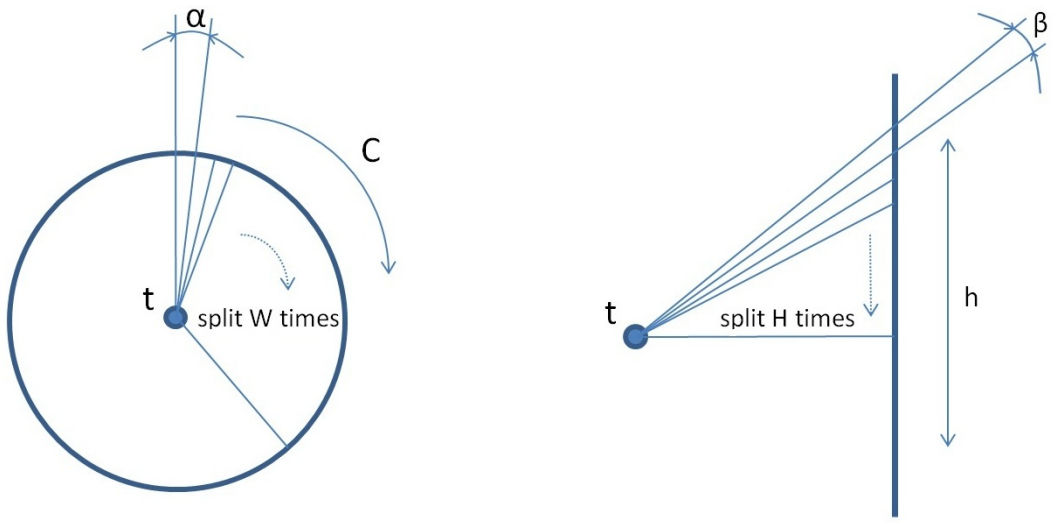


Figure 4.2: Angle resolution of a cylinder (in degrees). Left: angle resolution about the y-axis (α); right: angle resolution for the cylinder height (β); where t is the camera center

$$b = \frac{h}{H} \tag{4.2}$$

In other words for calculating the 3D-coordinates of the cylinder with regard to the panoramic map, the cylinder's C is split up into W parts around the y-axis for the x-coordinates and into H parts for the cylinder's height for the y-coordinates. The angle resolution about the y-axis in degrees is the angle that arises if a circle (a cylinder from a bird's eye perspective) is split up like a cake with very thin slices. For h it is similar, but in a vertical way (see Figure 4.2).

Every pixel of the panoramic map can be transformed into a 3D-vector originating from the camera center of the cylinder (0,0,0). The ray direction can be imagined as such a vector pointing in the direction of the camera orientation. To calculate the ray direction $\vec{r}$ the rotation matrix $\boldsymbol{R}$ is required. Assuming no gyroscope sensor or compass is used, the initial position of the camera is in the middle of the panoramic texture, where all rotations are 0. The resulting initial rotation matrix is the identity matrix. After initialization during the render cycles the rotation matrix will be calculated externally in the tracking process. In case of using sensors for estimating the initial orientation of the camera, the location of the camera frame depends on the camera orientation. The direction vector $\vec{d}$ is constantly pointing along the z-axis and in order to get the ray direction, the transpose of the rotation matrix is

multiplied with this vector.

$$\vec{r} = \mathbf{R^T}\ \vec{d} \tag{4.3}$$

For the calculation of the projection matrix the calibration matrix $\boldsymbol{K}$, the rotation matrix $\boldsymbol{R}$ and the camera location $\vec{t}$ are required. Assuming the phone's camera input is activated, initial values are set for the first cycles until the camera images are loaded from the phone's live preview feed. The calibration matrix $\boldsymbol{K}$ is a matrix with the camera calibration parameters such as the focal length (*fx, fy*) and the principle point (*px, py*). $\boldsymbol{P}$ is then calculated by multiplying the calibration matrix with the rotation matrix in consideration of the camera location.

$$\mathbf{K} = \begin{pmatrix} fx & 0 & px \\ 0 & fy & py \\ 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

$$\mathbf{P} = \mathbf{K}\ [\mathbf{R}|\vec{t}\,] \tag{4.5}$$

Since the camera is located in the center of the cylinder ($\vec{t}$*(0, 0, 0)*), the equation can be simplified to:

$$\mathbf{P} = \mathbf{K}\ \mathbf{R} \tag{4.6}$$

After preparing this information the data is sent to the fragment shader via uniforms. The coordinates of the input/output texture *(u,v)* are acquired from the vertex shader. In the fragment shader each fragment is mapped into cylinder space and checked if it falls into the camera image (backwards mapping). The cylinder coordinates $\vec{c}$*(x, y, z)* are calculated as follows:

$$c_x = sin(u\ a) \tag{4.7}$$

$$c_y = v\ b \tag{4.8}$$

$$c_z = cos(u\ a) \tag{4.9}$$

a and b are the angle resolutions as given in Formula 4.1 and 4.2.

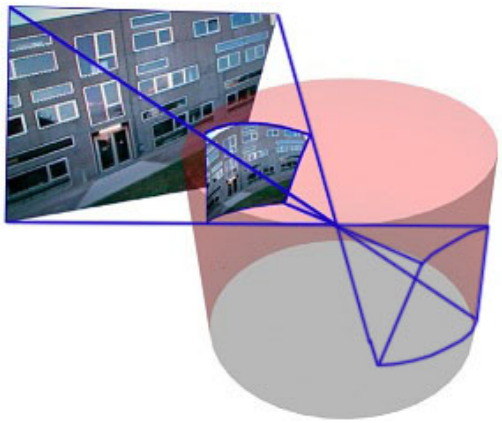


Figure 4.3: Projection of the camera image on the front and on the back of the cylinder (adopted from [Wagner10])

When projecting a camera image on the cylinder it is actually projected twice (once on the front-side and once on the back-side that is flipped). This is because the cylinder behaves like a pinhole camera, whereas the image in the front represents the image plane in front of the lens and the plane behind the lens is where the photo is captured (see Figure 4.3). To avoid mapping the image two times a check q whether the cylinder coordinates are in the front of the camera or in the back is performed using Equation 4.10. If the check *($q < 0$)* fails, the color of the corresponding input texture coordinate will be copied to the current fragment.

$$q = r_x\ c_x + r_y\ c_y + r_z\ c_z \tag{4.10}$$

The next step is to calculate the image coordinates $\vec{i}\,(x,y,z)$ in camera space. Therefore the projection matrix $\boldsymbol{P}$ is multiplied with the 3D-vector transformed from the cylinder coordinates. As mentioned above this is possible, because the camera center is positioned at (0,0,0) and each coordinate of the cylinder can be transformed into a 3D-vector.

$$i_x = \mathbf{P_{0,0}}\ c_x + \mathbf{P_{1,0}}\ c_y + \mathbf{P_{2,0}}\ c_z \tag{4.11}$$

$$i_y = \mathbf{P_{0,1}}\ c_x + \mathbf{P_{1,1}}\ c_y + \mathbf{P_{2,0}}\ c_z \tag{4.12}$$

$$i_z = \mathbf{P_{0,2}}\ c_x + \mathbf{P_{1,2}}\ c_y + \mathbf{P_{2,0}}\ c_z \tag{4.13}$$

To get the image point the homogenous coordinates are converted to image coordi-

nates.

$$x = \frac{i_x}{i_z} \tag{4.14}$$

$$y = \frac{i_y}{i_z} \tag{4.15}$$

After rounding the result to integral numbers the coordinates can be checked if they fall into the camera image. If this test fails, the color of the corresponding input texture coordinate will be copied to the current fragment again. If the test succeeds the color of the corresponding camera texture coordinate will be copied to the current fragment.

Without optimization this procedure is performed for all the fragments of the output texture. For a 2048x512 pixels texture resolution and therefore 2048x512 fragments every operation done in the shader will be executed over one million times. Even when discarding the shader program as soon it is known that the current fragment does not fall into the camera image, a lot of redundancy originates, due to the values for the checks have to be calculated.

4.3.2 Shader Optimization

Since mapping a camera image onto a panoramic map updates only a small region of the panoramic image, the shader program should not be executed for every fragment. Instead only the area where the camera image is mapped needs to be passed to the shader. To reduce the size of this area, the coordinates of the estimated camera frame, calculated in the tracking process, are used to create a bounding-box. The minimal and maximal coordinates of the bounding-box are then forwarded to a scissor test, where only the area that passes the test is passed to the shader (see Figure 4.4). This reduces the maximal number of shader runs from about 1,000,000 (2048x512 pixels) to about 75,000 (320x240 pixels), which is equivalent to a reduction in computational complexity to about 7.5 % over a naive implementation.

A second optimization step is to focus only on newly mapped fragments to further reduce the computing costs. Only those fragments should be mapped that were not mapped before. Assuming a panoramic image is tracked in real-time the frame is mapped about 25 times per second. If the camera is not moved too fast, only a very small area is new in the current frame. To achieve this reduction, newly updated cells

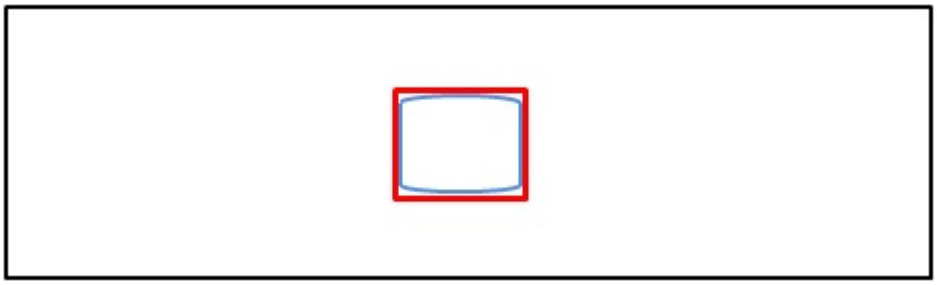


Figure 4.4: Black: panoramic map; blue: estimated camera frame; red: bounding-box that surrounds the camera frame an will be cut out by the scissor test

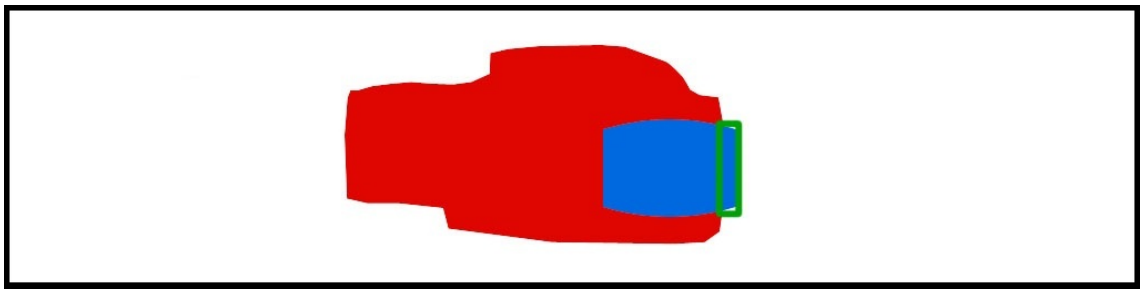


Figure 4.5: Red: mapped area; blue: current frame; green: small update region that is cut by the scissor test and passed to the shader. The additional optimization approach saves comptation costs.

that are already calculated by the tracker, are used. Each cell consists of an area of 64x64 pixels. If the cell is touched by the current tracking update, the coordinates are used to calculate another bounding-box around those cells. Then the intersecting area of the bounding-box of the whole camera image and the cell-bounding-box is cut again by the scissor test and passed to the shader as the new mapping area (see Figure 4.5).

Employing this optimization step does not necessarily reduce computational costs, because it directly depends on the movement of the camera. The update area can grow larger if the rotation of the camera results in a diagonal movement within panoramic space. Similarly, the update areas might become larger if the camera is rotated about the z-Axis. If more update areas come up at different locations the bounding-box can stay nearly the same size as in the approach described before, even if they are very small as shown in Figure 4.6.

Nevertheless processing only the newly mapped areas can reduce the number of shader runs significantly, since in more frequent cases only one small update area appears. The second optimization step is an additional improvement to the one

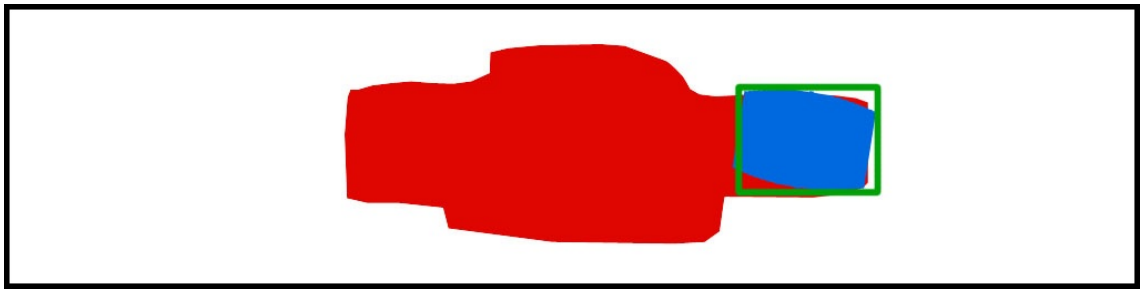


Figure 4.6: Red: mapped area; blue: current frame; green: big update region that is cut by the scissor test and passed to the shader. The additional optimization approach does not save a lot of computation costs.

calculating a bounding-box around the camera frame.

Using this mapping structure it is also important to consider the previous frame. Ignoring it could lead to unmapped gaps in the panoramic image due to the framebuffer-switch as described in the following example:

After a frame is mapped a framebuffer switch follows. The mapped frame is now in the input texture, but is not yet mapped in the current output texture. If a user moves the mobile phone too quickly or changes the direction of the camera movement, the newly calculated bounding-box might not include parts of the area of the previous frame and the mapped pixels of the input texture will not be copied to the output texture. The same problem can occur if the tracking process fails. This effect can be visualized as flickering in the panoramic preview.

To avoid these unmapped gaps the new mapping area has to be extended by including the coordinates of the previous render cycle for the final mapping-bounding-box.

The following example is a fragment shader snippet for determining whether a fragment falls into the camera frame or not, that is calculated for each fragment after passing the scissor test:

```
1  vec4 tempColor; // corresponding color value of the input texture
2  tempColor = texture2D(s_texture_bg,v_texCoord_bg/vec2(panoW,panoH));
3
4  // x- and y-cooridnate of the panoramic map
5  float x = v_texCoord_bg.x;
6  float y = v_texCoord_bg.y - panoH / 2.0;
7
8  // get cylinder coordinates
9  float xC = sin(x * angleResX);
10 float yC = y * angleResY;
```

```
11 float zC = cos(x * angleResX);
12
13 // check if frame is in front of the camera
14 float check = (rayDir.x * xC  +  rayDir.y * yC  +  rayDir.z * zC);
15
16 if(check < 0.0) {
17   // project
18   vec3 imgCoord;
19   imgCoord.x = P[0][0]*xC + P[1][0]*yC + P[2][0]*zC;
20   imgCoord.y = P[0][1]*xC + P[1][1]*yC + P[2][1]*zC;
21   imgCoord.z = P[0][2]*xC + P[1][2]*yC + P[2][2]*zC;
22
23   // image point
24   float X = imgCoord.x / imgCoord.z;
25   float Y = imgCoord.y / imgCoord.z;
26
27   // round to integral coordinate values
28   vec2 imageCoords = vec2(floor(X + 0.5), floor(realCamH - Y - 0.5));
29
30   // check if image point falls into the camera image
31   if((imageCoords.x >= 0.0 && imageCoords.x < realCamW) &&
32      (imageCoords.y >= 0.0 && imageCoords.y < realCamH)) {
33     vec4 camColor; // copy color value from the camera image
34     camColor = texture2D(s_texture_cam,imageCoords/vec2(camW,camH));
35     color = camColor;
36   } else { // copy the color from the input texture
37     color = tempColor;
38   }
39 } else { // copy the color from the input texture
40   color = tempColor;
41 }
42 gl_FragColor = color;
```

Chapter 5

GPU-based Mapping for advanced applications

The main goal of this work is to exploit the advantages of parallel processing on the GPU. The possibility to use shader for image processing allows to perform approaches that are extremly costly to compute on CPUs, however can be realized with little computational effort on the GPU. Such approaches with regard to the mapping process are for example image refinement methods that require pixel blending, clearing certain areas or enlarging the amount of pixels to be rendered.

In the following sections approaches are discussed to enhance the quality and usability of panoramic images.

5.1 Wiping

A very powerful feature is the possibility to wipe out areas in the panoramic images in real-time. Taking a panoramic photograph is a time consuming process compared to taking a normal photograph. Panoramic images happen to contain unwanted areas like persons or cars that cover an essential part of the scene. To remove these unwanted spots the panoramic image can be edited in real-time. For example, by specifying an area in the preview image of a panoramic map, the coordinates might be passed to the shader and the region around that coordinate is cleared and marked as unmapped. A new frame arriving can cover those cleared areas and fill the empty spots with color information again.

A possible implementation of this feature is a simple swipe operation on a touch screen. In such an implementation, the area around the coordinates that has been marked to clear is defined to be circular with a radius of N pixels. The program simply passes the coordinates to the fragment shader. There the clearing area is calculated using the dot product of the euclidean distance between the current fragment

Figure 5.1: Circular white spots have been cleared while taking the panoramic image

coordinate $\vec{t}$ and the marked wiping coordinates $\vec{w}$.

$$(\vec{t} - \vec{w}) \cdot (\vec{t} - \vec{w}) < (N^2) \tag{5.1}$$

If the condition is true and the wiping coordinate lies within the euclidean distance, the pixel that is currently processed by the fragment shader can be cleared. This approach can also be computed in a CPU-based mapping process, but the advantage of the GPU-based wiping is that it runs in real-time.

The shader implementation for a wiping process is realized as follows:

```
1 vec2 click = v_texCoord_bg-wipeCoords;
2 if(dot(click, click) < N*N)
3 {
4   color = vec4(1.0, 1.0, 1.0, 1.0);
5 }
```

5.2 Image Refinement

A significant problem while taking panoramic images in real-time is the changing exposure time of mobile phones' cameras, due to increasing or decreasing intensity of the incoming light. Programmers can not manually access and control the exposure time according to the camera orientation and the light source. Moving the camera towards a light source significantly darkens the input image of the camera's live preview feed. Moving the camera away from the light source will brighten the input image in an unproportional way. The artifacts that arise due to the diverging exposure time are sharp edges between earlier mapped regions and newly mapped camera images as seen in Figure 5.2.

The field of taking panoramic images and handling the differences of the exposure time is widely discussed as described in Section 2, but the problem statement differs

Figure 5.2: Sharp edges in homogenous areas due to diverging exposure time

from this approach. As Wagner et. al. [Wagner10] mentioned, several approaches dealing with the exposure problem do not map and track in real-time or need some pre- and/or post-processing to create a seamless panoramic image. Additionally most of the other approaches require a lot of memory since they use the taken images for post-processing and therefore have to store them. Using a GPU-based mapping approach however, we can directly employ shading and blending effects right while the panoramic image is recorded. No additional image information has to be stored on the device. Using the attributes of a GPU, the postprocessing steps therefore vanish and becomes an active part of the real-time capturing of a panorama for certain approaches.

Several different approaches are investigated in the following sections to enhance the image quality.

5.2.1 Brightness Offset Correction

One way to manually correct the differences in brightness values of the current camera image is to find matching points in the panoramic image and the camera image and calculate their brightness difference from the color data. The average offset of these differences is then forwarded to the shader and considered in the mapping process.

To calculate the brightness offset of matching points the approach implemented in the thesis of Degendorfer [Degendorfer10] is revised. Degendorfer calculates the brightness offset for the feature points found by the tracker (see Figure 5.3). This solution is not ideal, however, as the best areas for comparing brightnesses are homogenous regions rather than corners. The advantage of this approach is that it can be performed at almost no additional computational overhead, since the tracker inherently provides the matches and the actual pixel values are just compared.

Figure 5.3: Brightness offset correction calculated from feature points [Degendorfer10]

To avoid using misleading brightness values at feature points, a grid (e.g. 16x12) is laid over the camera image. The coordinates are forward-mapped onto the panoramic map to get the corresponding coordinates in the panoramic image. These coordinates are stored in a texture and forwarded to the fragment shader. In the shader the according color value and brightness offset is calculated, if the fragment of the panoramic map is already mapped. The resulting average offset is considered in the process of mapping new pixels. This solution requires some computational effort, since the forward-mapping of the camera coordinates onto the panoramic map is a costly operation, which has to be calculated for each grid point. The advantage of this approach is that the brightness differences are not only measured at corner points.

Adding or subtracting an offset of a color value can lead to color values below 0 and above 255. Two different methods of dealing with this circumstance can be realized. The first one truncates all values lower than 0 (negative values) to 0 and all values higher than 255 to 255. This leads to very dark and very bright areas depending of the location of the light source. The second method stores the offset in the alpha channel of the panoramic map, since it is not used otherwise. Occupying the first 127 values for the positive offset and the second 127 values for negative offset values, the color of each pixel can be modified during the saving process of the panoramic image (post processing) and therefore costs no additional computation time during the mapping process.

For the second method using an extended range for the storing the color value, a tone mapping is mandatory. Otherwise the panoramic image can suffer from strong contrast reduction and colors will appear grayish.

5.2.2 Gamma Correction

Images that are gamma encoded can store tones more efficiently. The reason therefore is that cameras do not perceive light the same way the human eyes do. Too few bits are used to describe darker tones (where the camera is less sensitive) and too many bits are used for highlights that cannot be differentiated by the human eye [Plataniotis00]. To compensate this effect a color correction regarding the gamma encoding is realized using the formula from the ITU-R BT.709 standard [ITU-R90].

For a red value R, green value G and blue value B, where R, G and $B < 0.018$

$$R' = 4.5\ R \tag{5.2}$$

$$G' = 4.5\ G \tag{5.3}$$

$$B' = 4.5\ B \tag{5.4}$$

and for R, G and $B \geq 0.018$

$$R' = 1.099\ R^{0.45} - 0.099 \tag{5.5}$$

$$G' = 1.099\ G^{0.45} - 0.099 \tag{5.6}$$

$$B' = 1.099\ B^{0.45} - 0.099 \tag{5.7}$$

is used to calculate the gamma corrected color.

5.2.3 Pixel-Blending

Blending the camera image with the panoramic image in the mapping process is a way to smoothen sharp transitions of different brightness values. To achieve smoother transitions several different blending approaches are investigated.

All blending operations can be combined with the brightness offset correction and the gamma correction.

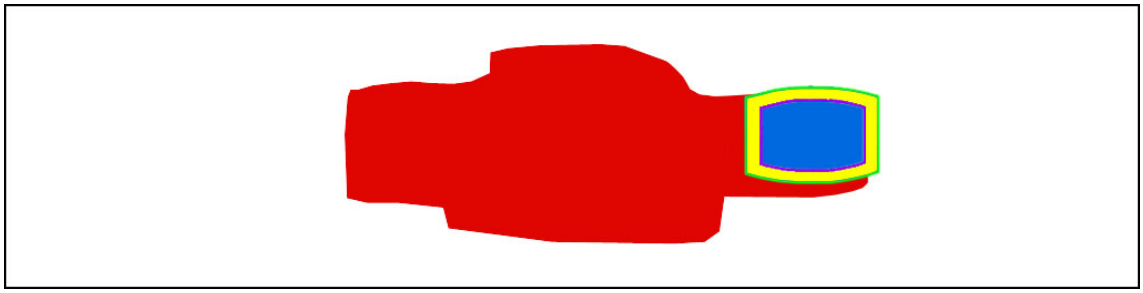


Figure 5.4: Linearly blending the camera image with the panoramic image in the frame area (yellow) between the outer (green) and the inner (purple) blending frame.

5.2.3.1 Blending a Frame Area of the Camera Image

Since the camera image does not cover 100 % of the already mapped panoramic map, not every pixel can be blended. The color values of newly mapped pixels have to be drawn as they appear in the camera image or they would be blended with the initial white background color. To avoid having sharp edges at the border to the newly mapped pixels, only a frame area represented by an inner and an outer frame is blended as shown in Figure 5.4. Pixels at the image border (outer frame) are taken from the panoramic map. A linear blending operation is used in the area between the frames along the direction of the normal to the outer frame. The region inside the blending frame is directly mapped from the camera image.

To avoid blending the frame with unmapped white background color, new pixels are mapped without blending directly from the camera image.

Blending two images using the fragment shader is a computationally cheap operation and can easily be applied to the naive form of pixel mapping. However, the pixel-blending requires the optimization method where the whole camera image is updated in every frame, as the area of the whole camera frame is required for the blending process.

5.2.3.2 Blending the Running Average of the camera image

This blending approach blends the whole camera image with the panoramic map. Newly mapped pixels that would be mapped with the white background color are

completely mapped from the camera image. To avoid having visible edges at the border to the newly mapped pixels, the mapping process is realized using the average of the panoramic map and the camera image. The total number of times a pixel has been mapped, is stored in the alpha value and integrated in the average calculation. This means that a pixel that has not been mapped before, is simply copied from the camera image, as mentioned above. The second time the same pixel is mapped the average of the before mapped color value and the new color value from the current camera frame is mapped. In the next mapping cycle of this pixel the value of the camera image is integrated in the average calculation using the cumulative running/moving average ca:

$$ca_{i+1} = \frac{x_{i+1} + i\ ca_i}{i+1} \tag{5.8}$$

x is the value of the camera image and i is the total number of times the pixel has already been mapped, fetched from the alpha channel.

Since the alpha value consists of eight bit each pixel can be updated a maximum time of 255.

5.2.3.3 Blending Values in HSV-Color Space

A similar way of blending the color values to reduce the difference of brightnesses between the current camera image and the panoramic map is to blend the brightness values. The RGB-color space (red, green, blue) is converted into the HSV-color space (hue, saturation, value). The value of the each pixel to be mapped is blended with the value of the pixel of the panoramic image. The blending approach is combined with the frame area blending and blending the whole camera image using the cumulative running average.

The difference between the two color models is that the RGB-model is an additive color model with its additive prime colors and the HSV-model is based on perceptual variables [Smith78]. The HSV model is represented in cylindrical form and often conveniently represented by a hexcone. The primary color red is at 0 degrees of the cylinder, followed by green at 120 degrees and blue at 240 degrees. The colors in between are mixed colors. The hue changes around the y-axis, the saturation increases from the center of the cylinder to the outside and the value lies on the vertical axis [Plataniotis00].

The conversion from the RGB-model to the HSV-model and vice versa is shown in the following shader snippet:

```
vec3 HSVtoRGB(vec3 HSV)
{
  vec3 RGB = vec3(HSV.z,HSV.z,HSV.z);
  float hi = floor(HSV.x * 6.0);
  float f = HSV.x * 6.0 - hi;
  float p = HSV.z * (1.0-HSV.y);
  float q = HSV.z * (1.0-HSV.y*f);
  float t = HSV.z * (1.0-HSV.y*(1.0-f));

  if(HSV.y != 0.0)
  {
    if (hi == 0.0 || hi == 6.0) { RGB = vec3(HSV.z, t, p); }
    else if (hi == 1.0) { RGB = vec3(q, HSV.z, p); }
    else if (hi == 2.0) { RGB = vec3(p, HSV.z, t); }
    else if (hi == 3.0) { RGB = vec3(p, q, HSV.z); }
    else if (hi == 4.0) { RGB = vec3(t, p, HSV.z); }
    else { RGB = vec3(HSV.z, p, q); }
  }
  return RGB;
}

vec3 RGBtoHSV(vec3 RGB)
{
  vec3 HSV = vec3(0.0,0.0,0.0);
  float minimum = min(RGB.r, min(RGB.g, RGB.b));
  float maximum = max(RGB.r, max(RGB.g, RGB.b));

  if(maximum == minimum) { HSV.x = 0.0; }
  else if (maximum == RGB.r)
  { HSV.x = ((RGB.g - RGB.b) / (maximum - minimum))/6.0; }
  else if (maximum == RGB.g)
  { HSV.x = (2.0 + (RGB.b - RGB.r) / (maximum - minimum))/6.0; }
  else if (maximum == RGB.b)
  { HSV.x = (4.0 + (RGB.r - RGB.g) / (maximum - minimum))/6.0; }

  if( HSV.x < 0.0) { HSV.x += 1.0; }

  if(maximum == 0.0) { HSV.y = 0.0; }
  else { HSV.y = (maximum - minimum) / maximum; }

  HSV.z = maximum;
  return HSV;
}
```

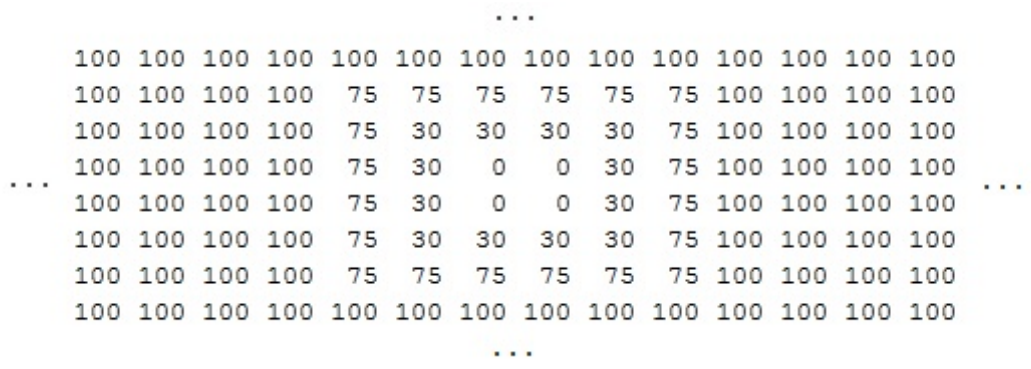


Figure 5.5: States of the cell-grid texture containing blending factors

5.2.3.4 Blending/Fading from Completed Cells

In this approach pixels that have been mapped more often and are located further in the center of the panoramic map, are blended with less intensity than the newly mapped pixels at border of the mapped area. New pixels are directly mapped from the camera image, whereas the further a pixel lies in an already mapped region the less influence the color values of the camera image have. To achieve a correct blending, completed cells of the tracking process are used to get the already mapped areas of the panoramic map. This requires a split of the panoramic map into cells (e.g. 32x8) as shown Figure 3.2. A texture with the size of the cell grid is created to store a state value for each cell, which describes how strong the pixels should be blended. The texture is initialized with a blending factor that directly maps the camera frame. The first image is simply copied on the panoramic map. For a new frame, a few cells are already completed and the state of the cell-grid texture is updated and the blending factor reduced. If a state value in the texture is surrounded by lower or equal states it decreases its value again. If the same thing happens to a state that has already been decreased two times before, one could disable the blending by setting the blend factor to 0 (see Figure 5.5). The values shown in Figure 5.5 are sample values and can be adjusted to ones needs. However a more or less linear blending curve with a steeper section between the first and the second reduction of the blending factor turned out work best.

The texture is passed to the vertex shader and the respective blending factor, taken from the texture is interpolated and forwarded to the fragment shader. Therefore the vertex shader has to have the same grid of vertices of the panoramic map as cell-grid texture. In the fragment shader the interpolated blending factor is integrated in the mapping process.

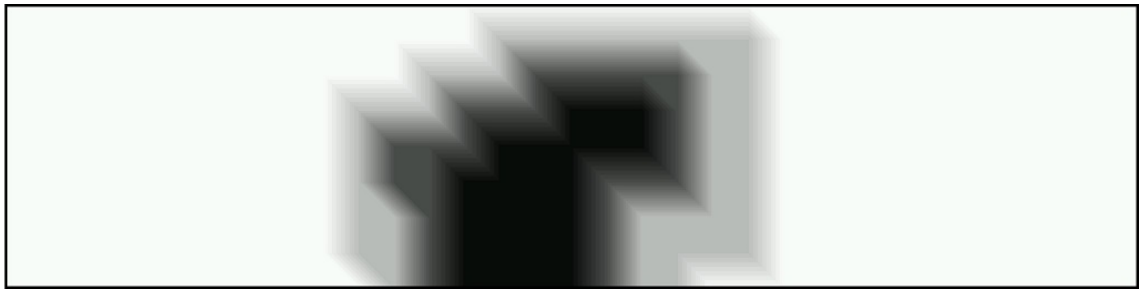


Figure 5.6: The blending map with the interpolated blending factors taken from the cell-grid texture

The resulting blending map of this approach is shown in Figure 5.6.

5.3 Larger panoramic images

Mapping a panoramic image on a CPU in real-time is possible for medium-size panoramic images as shown in [Wagner10]. Increasing the panoramic map and the camera image resolution for real-time CPU-based mapping it will quickly meet its limits in computational power. The GPU-based mapping approach can handle larger texture sizes with a neglectable loss of render speed. Reducing the area passed to the fragment shader in an optimization step, the size of the panoramic map does not have much influence on the real-time frame rates. The camera image size would have more influence, however, the live preview feed of recent mobiles, which is about 640x480 pixels can still be rendered in real-time.

A limitation for the GPU-mapping is the limited texture size of a mobile phone's GPU. This problem can be avoided by splitting the panoramic texture into several parts.

Chapter 6

Evaluation and Experimental Results

In this chapter we describe the results of a variety of experiments using a large number of different scenarios. Most of the tests were conducted with previously recorded videos to allow comparing individual approaches directly. Although several field tests are conducted directly with a mobile phone, a steady test scenario is chosen to compare the individual improvements more objectively.

The evaluation is divided into three main sections. In the first section the image quality is tested by means of the image refinement approaches discussed in the previous chapter. The second test section compares the results of the refinement approaches in terms of robustness of the tracking process and in the third section the render speed performed for every approach is tested.

The test devices and their hardware specifications used for this evaluation are listed in the table below:

6.1 Panoramic image refinement

In the first evaluation step five videos are taken for every location. One during a sunny day (outdoor #1), one on a cloudy day (outdoor #2), one in the evening/-dawn light (outdoor #3), one indoors using artificial light (indoor #1) and one indoors with daylight (indoor #2). The location for the outdoor scenes is located at the University campus and for the indoor scenes a well illuminated room is chosen. For realizing the test with artificial light, the blends of the windows are

Device	**CPU**	**GPU**	**Operating System**
Samsung Galaxy S II	1.2GHz dual core	Mali-400MP	Android 2.3.5
LG Optimus 4x HD	1.5GHz quad core	Nvidia Tegra 3	Android 4.0.3
Samsung Galaxy S II	1.4GHz quad core	Mali-400MP	Android 4.0.3

Table 6.1: Hardware specifications of the devices used for the evaluation

closed.

As reference for every test case the CPU-mapped image is compared with the GPU-mapped results for improvements.

Outdoor Scenario #1: In this test a panoramic image is taken of a sunny outdoor scene. As mentioned before the most difficult process of seamlessly mapping the camera image in the panoramic map is to cover the brightness differences. Having the sun as a strong light source in the scene complicates it even more. However distinctive shadow structures enable the tracker to find corresponding points on otherwise homogenous regions.

Figure 6.1(a) is the reference image created by the PanoMT-application ([Wagner10]). Brightness differences are significantly visible. Even seams between consecutively mapped camera images are visible and artifacts appear in the lower region of the panoramic image.

Reducing the brightness differences with a modified brightness correction version described by [Degendorfer10] slightly reduces the differences in brightness between former mapped camera images and later mapped images, but emphasizes the brightness seams between consecutively mapped camera images as shown in Figure 6.1(b).

In Figure 6.1(c) the seams between the consecutively camera images as well as general differences in brightness are smoothed by the blending approach described in Section 5.2.3.1. The test is a combination of the brightness correction and the blending process. As a result of the smoothing of the seams the image gets a bit blurry, however it emphasizes the impression of one continuous image. Artifacts like lens flares are visible since in this approach the whole camera image is mapped every time. Approaches that only map new pixels usually do not suffer from these artifacts since lens flares do not appear at image borders very often. The gray area in the left half of the image originates from the brightening process of an almost black region, due to brightness correction.

The algorithm for image refinement that relies on completed cells of the panoramic map as described in Section 5.2.3.4 generates blurry panoramic images (6.1(d)), due to inaccuracies of the tracker. Since the camera image is only mapped directly at the border, edges or details are not blended and mapped on the very same locations and therefore appear blurry. Furthermore some seams stay visible, due to the limited blending cycles and lens flares appear because of blending the whole camera image.

Using gamma correction for image refinement reduces the perceived brightness dif-

(a)

(b)

(c)

(d)

Figure 6.1: Panoramic images of a sunny scene

ferences, but brightens the panoramic image too much. The general impression of the color intensities decreases and seams are still visible.

The grid brightness correction, where the brightness values are not taken from the feature points, but from a grid laid on the camera image (described in Section 5.2.1), does not deliver better results than the brightness offset calculated by [Degendorfer10] and requires additional computational costs. Therefore, especially for combinations with other refinement methods, Degendorfer's approach is used.

The approach using the frame-blending method for blending HSV-Values creates similar results compared to the common frame-blending approach, except that the panoramic image appears a bit grainy in homogenous areas and and it takes additional computation power to generate the panoramic image.

Calculating the running average of the pixels' color values results in a blurry image. The reason for that is the same as for the completed cell approach explained in this section before.

Outdoor Scenario #2: In the second outdoor test a panoramic image is taken of a cloudy scene. The difference to the sunny scene is that the sun as light source is significantly less visible. However larger homogenous regions appear, due to the absence of shadows and the lesser intensity of colors.

Figure 6.2(a) is the reference image created by the PanoMT-application ([Wagner10]). Significant brightness differences are visible, similar to the result image (see Figure 6.1(a)) of the sunny scene. Also strong mapping artifacts are visible, especially in the middle of the left half of the panoramic image.

Similarly to the sunny scene result the brightness correction reduces the differences in brightness, but seams are still visible as shown in Figure 6.2(b). The artifacts in the left image half stay unchanged like in the CPU-mapping approach.

Frame-blending (see Figure 6.2(c) reduces most of the seams, but the artifacts created by slightly inaccurate tracking result in blurry areas. Due to the covered light source, there is no unwanted additional gray region created by brightness corrections, in contrast to the sunny scene.

Using the completed cell image refinement approach brightness differences are partially visible and the image is very blurry (see Figure 6.2(d)).

The general impression of the decreasing color intensities when adding a gamma correction to the algorithms appears less than in the sunny scene, due to the gray

(a)

(b)

(c)

(d)

Figure 6.2: Panoramic images of a cloudy scene

weather conditions. However the image quality is not increased significantly.

Also the other image refinement approaches described in Section 5.2 cannot achieve better results concerning the visual image quality and produce similar outputs than in the sunny outdoor scene.

Outdoor Scenario #3: In the third outdoor test a panoramic image is taken in the evening with dawn light. The difference to the sunny and the cloudy scene is that there is no light source visible. Larger homogenous regions appear, due to the absence of shadows, the lesser intensity of colors and dawn light exacerbates the tracking to find matching points.

Figure 6.3(a) is the reference image created by the PanoMT-application ([Wagner10]). Significant seams of brightness differences are visible, similar to the result images of the sunny and the cloudy scene (see Figure 6.1(a) and 6.2(a)). Some artifacts appear in the lower area of the image.

The brightness correction removes the artifacts and reduces the strength of the brightness differences. However it generates more visible seams between consecutively mapped camera images as seen in Figure 6.3(b).

The blending approach reduces the number of seams significantly, but some parts of the image appear a bit more blurry (see Figure 6.3(c)). Even if a few regions of the image are differently illuminated the general impression of the panoramic image is improved. Since no sun is in the image, gray areas possibly generated from the brightness correction are not visible.

The completed cell blending cannot remove all seams and as in the other scenes the image appears more blurry than using other algorithms as shown in Figure 6.3(d).

Due to the dawn light of the evening, the decreasing color intensities when adding a gamma correction to the algorithms appear less pronounced than in the sunny scene. As seen in the case of cloudy weather, the image quality is not increased significantly.

Also the other image refinement approaches behave similarly to the outdoor scenes discussed above and are not further discussed for this test.

Indoor Scenario #1: In this test a panoramic image is taken from an indoor scene with artificial light. Compared to the outdoor environment the scene is much closer to the camera location and translational movement of the camera caused by an inexperienced user has a stronger effect on panoramic images than more distant

(a)

(b)

(c)

(d)

Figure 6.3: Panoramic images of a evening scene

scenes. Therefore visual artifacts are created more easily. The artificial light generates an evenly illuminated environment that reduces the significance of brightness differences.

Figure 6.4(a) is the reference image created by the PanoMT-application ([Wagner10]). Significant seams of brightness differences are visible, even if the room is illuminated homogenously. Some artifacts appear in the lower area of the image and cuts arise, due to the imprecise tracking of close objects.

Since the differences in brightness are less significant for the indoor test using artificial light, the brightness correction does not achieve a better result in contrast to the outdoor tests. The brightness correction effort is neglectable and in addition seams occur between consecutively mapped camera frames as shown in Figure 6.4(b). Artifacts generated in the CPU-mapped image are not visible using GPU-mapping.

The frame-blending approach creates a very good result concerning the brightness differences. However the image appears a bit more blurry and a slight deformation occurs, due to imprecise tracking of close objects (see Figure 6.4(c)).

In Figure 6.4(d) the result of the completed cell blending algorithm is shown. The panoramic image is significantly more blurry than the frame-blending approach. Furthermore the deformation is stronger and a few brightness seams and cuts are still visible.

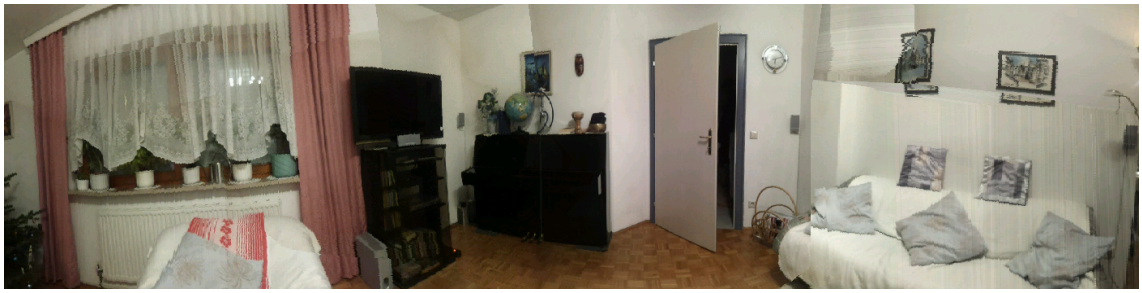
Adding a gamma correction to the panoramic image allows the user to see reflections on specular surfaces, but brightens the whole image too much and therefore decreases the color intensities.

The grid brightness correction does not have a significant effect on the image quality and the HSV-blending achieves a worse result than the frame-blending approach, but has higher computational costs.

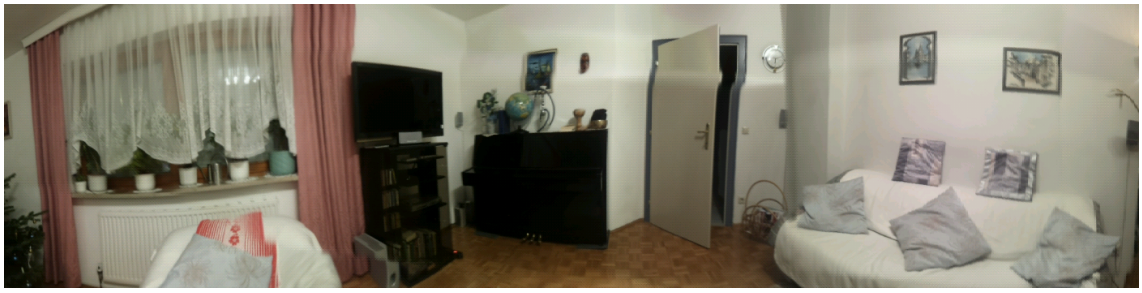
Indoor Scenario #2: In the second indoor test a panoramic image is taken using natural light. The difference to the artificial light source is that not all areas of the image are evenly illuminated. Equal to the previous indoor test, the short distance of the camera to the scene objects is a problem for the tracker. Significant brightness differences harden the process of taking a panoramic image even more.

Figure 6.5(a) is the reference image created by the PanoMT-application ([Wagner10]). Due to strong brightness differences and close scene objects, significant seams and cuts are visible. Additionally render artifacts appear in the lower area of the panoramic

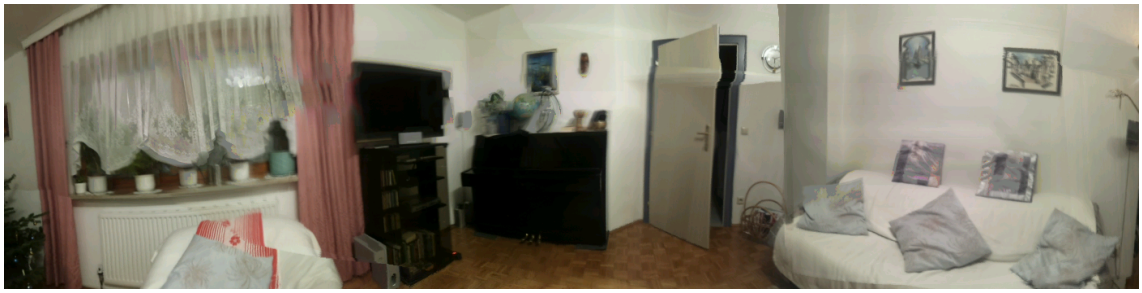
(a)



(b)



(c)



(d)

Figure 6.4: Panoramic images of an indoor scene with artificial light

image.

Similar to the other tests the brightness correction decreases the general brightness differences, but generates additional seams between consecutively mapped frames as shown in Figure 6.5(b). Artifacts that are visible in the CPU-mapped result image are eliminated.

The strongly visible seams due to brightness differences are smoothed, but still partly visible. Similar to the result images of the other tests, the image appears a bit more blurry and deformation occurs, due to imprecise tracking of close objects (see Figure 6.5(c)).

The completed cells algorithm generates a blurry image with brightness seams and deformations (see Figure 6.5(d)). Additionally the image has a significant red cast. A tinge of red is also noticeable in the other tests of the indoor scene with natural light, but not as strong as in the test using the completed cell image refinement approach.

As in the indoor test using artificial light, adding gamma correction shows more details in reflecting areas, however the general impression is too bright and the color intensity is decreased. The grid brightness correction does not have a significant effect on the image quality and the HSV-blending achieves a worse result than the frame-blending approach with higher computational costs.

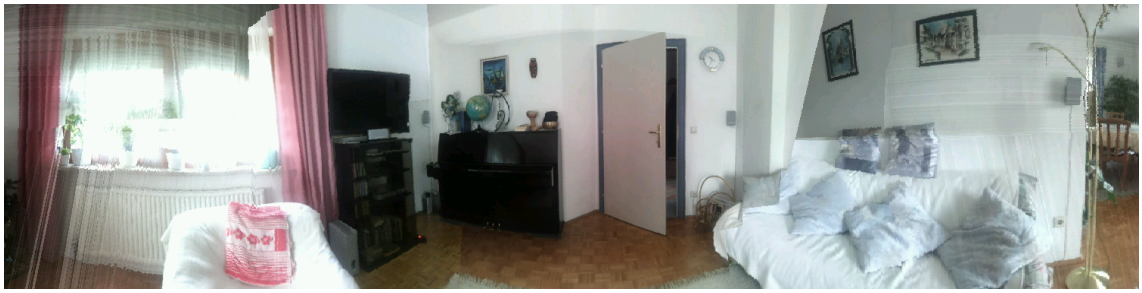
6.2 Robustness

The results of each test run are not only tested visually, but also by forwarding the GPU-mapped images to the tracker and calculating matching points for the current camera image. Subsequently the amount of key points found is compared with the number of key points found using the CPU-mapped image. Getting a higher number of matching points increases the tracking robustness and confirms an improvement to the existing PanoMT-application.

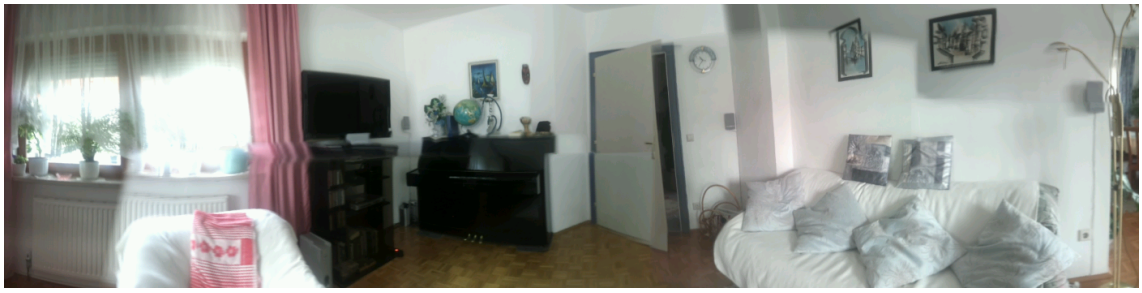
The following tables 6.2 - 6.6 show the individual tests listed with the results of the found key points and the number of matches. For each frame the key points and their matches are stored and the average value of all found feature points and matches are taken for comparison. An image refinement approach that reaches a higher score of averagely found key points and reaches the maximum of 80 matches, is considered to be more robust than approaches with a lower score.

The most robust tracking for the sunny outdoor scene is achieved by the grid bright-

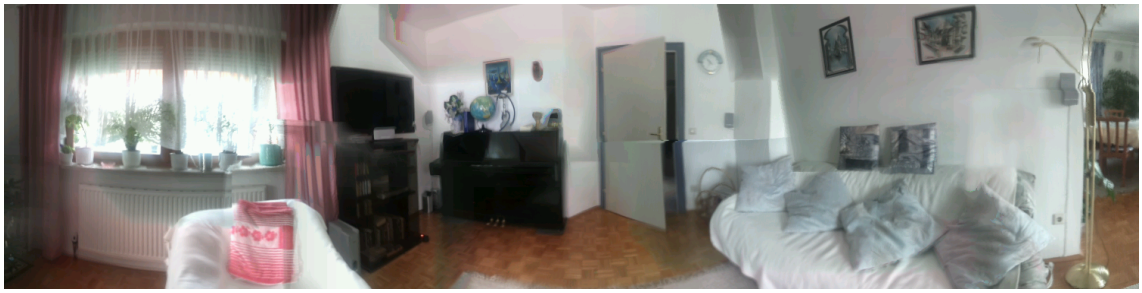
(a)



(b)



(c)



(d)

Figure 6.5: Panoramic images of an indoor scene with natural light

Approach	⌀ Matches	⌀ Key Points
CPU		
Standard Mapping	80.00	1000.30
GPU		
No Refinements	80.00	1050.18
Brightness Correction from Feature Points	80.00	1053.41
Gamma Correction	80.00	1025.37
Grid Brightness Correction	80.00	1062.92
Running Average	78.56	753.39
Frame Blending	73.95	1028.54
Frame Blending + Brightness Correction	78.03	1030.62
HSV-Blending	71.29	1042.67
Completed Cell	77.48	1036.53
Completed Cell + Brightness Correction	78.93	1026.57

Table 6.2: Average of found key and matching points for each refinement approach (sunny)

ness correction image refinement algorithm, with an average of 1062.92 key points found per frame where the maximum number of 80 matches are found. In general all GPU-mapping algorithms reached a higher score than the standard CPU-mapping implemented in the PanoMT-application, except the running average algorithm, which seems to create panoramic pictures that are too blurry for tracking (see Table 6.2).

For the cloudy outdoor scene the most robust tracking is achieved by the brightness correction, where the offset is gained from feature points, with an average of 811.74 key points found per frame and a maximum of 80 matches. In general all GPU-mapping algorithms that do not map the whole camera image, but only the unmapped pixels for each frame, reach higher scores than the standard CPU-mapping implemented in the PanoMT-application. For the other algorithms the tracker suffers from blurry panoramic images and cannot find as much matching points (see Table 6.3).

For tracking the outdoor scene in the evening the most robust image refinement algorithm appears to be the standard GPU-mapping approach with additional gamma correction with a score of 773.78 key points and a maximum of 80 matches per frame. In this test only a few approaches achieve a better score than the CPU-mapping approach. For taking a panoramic image with dawn light, image refine-

Approach	⌀ Matches	⌀ Key Points
CPU		
Standard Mapping	80.00	792.54
GPU		
No Refinements	80.00	800.72
Brightness Correction from Feature Points	80.00	811,74
Gamma Correction	80.00	729.36
Grid Brightness Correction	80.00	809.32
Running Average	79.98	443.04
Frame Blending	80.00	589.52
Frame Blending + Brightness Correction	80.00	572.80
HSV-Blending	80.00	654.30
Completed Cell	80.00	474.59
Completed Cell + Brightness Correction	80.00	625.83

Table 6.3: Average of found key and matching points for each refinement approach (cloudy)

Approach	⌀ Matches	⌀ Key Points
CPU		
Standard Mapping	80.00	607.02
GPU		
No Refinements	80.00	605.75
Brightness Correction from Feature Points	80.00	595.35
Gamma Correction	80.00	649.32
Grid Brightness Correction	80.00	616.52
Running Average	80.00	518.69
Frame Blending	80.00	485.37
Frame Blending + Brightness Correction	80.00	481.65
HSV-Blending	80.00	488.13
Completed Cell	80.00	490.13
Completed Cell + Brightness Correction	80.00	493.59

Table 6.4: Average of found key and matching points for each refinement approach (evening)

Approach	⌀ Matches	⌀ Key Points
CPU		
Standard Mapping	80.00	334.63
GPU		
No Refinements	80.00	341.76
Brightness Correction from Feature Points	80.00	348.15
Gamma Correction	80.00	444.45
Grid Brightness Correction	80.00	346.79
Running Average	79.94	284.70
Frame Blending	77.43	364.24
Frame Blending + Brightness Correction	77.87	342.78
HSV-Blending	77.71	363.02
Completed Cell	80.00	443.71
Completed Cell + Brightness Correction	80.00	440.24

Table 6.5: Average of found key and matching points for each refinement approach (indoor with artificial light)

ments that only update newly mapped pixels seem to be more accurate (see Table 6.4).

The indoor environment illuminated with artificial light appears to be tracked most robustly by the mapping approach using gamma correction with a score of 444.45 key points and a maximum of 80 matches per frame. All image refinement algorithms achieve a higher score than the CPU-mapped version except the running average algorithm, which seems to create too blurry panoramic images for tracking (see Table 6.5).

The naturally illuminated indoor scene is most robustly tracked by the standard CPU-mapping with a score of 450.83 key points and a maximum of 79.98 matches per frame. The grid brightness correction approach achieves a higher score of 484.35 key points, but only 72.36 matches can be found (see Table 6.6).

In general the robustness of tracking is higher for approaches that update only pixels that have not been mapped before. Comparing the results of those GPU-mapping approaches to the approach developed by Wagner et. al., the GPU-mapping approaches achieved higher results. The approaches using blending for image refinement generate images that are a bit blurry and therefore less tracking points and their matches can be found.

Approach	⌀ Matches	⌀ Key Points
CPU		
Standard Mapping	79.98	450.83
GPU		
No Refinements	72.64	436.95
Brightness Correction from Feature Points	74.42	445.00
Gamma Correction	71.09	465.45
Grid Brightness Correction	72.36	484.35
Running Average	73.38	337.40
Frame Blending	72.22	407.64
Frame Blending + Brightness Correction	76.14	415.42
HSV-Blending	71.99	402.47
Completed Cell	68.75	225.76
Completed Cell + Brightness Correction	68.54	225.02

Table 6.6: Average of found key and matching points for each refinement approach (indoor with natural light)

6.3 Render Speed

The speed tests discussed in this section measure the averagely rendered frames per second for each image refinement approach and for different panoramic mapping sizes. For calculating the frame rate the first 50 frames are dismissed and then the average of the next 50 frames is taken to determine the speed of the current image refinement approach. Each test is run three times for each refinement approach and mobile phone. The average of the results is taken as the render speed result. For testing the speed differences for different panoramic mapping sizes, two resolutions are chosen. A lower and standard texture resolution of 2048x515 pixels and a higher texture resolution of 4096x1024 pixels are realized for this test.

The tests are realized with three different testing devices to show possible differences of render speed in different render sections. Such sections are the calculation on the CPU-side (data preparation for the shader) and GPU-side (shader runs). Since the frame rate is dependent of the stronger bottleneck of these sections, one can see if the application has free resources on the CPU- or the GPU-side. The testing devices are listed in Table 6.1.

For algorithms that only map pixels that have not been mapped before, the area that passes the scissor test is set to 0, if the current camera image contains no new information to map onto the panoramic map. This increases the FPS during

Approach	SGS2 FPS (low/high)	LG-4xHD FPS (low/high)	SGS3 FPS (low/high)
No Refinements	27.50 / 25.67	27.55 / -	22.46 / 21.15
Brightness Correction from Feature Points	27.20 / 25.08	26.55 / -	21.94 / 20.77
Gamma Correction	27.32 / 25.23	26.78 / -	22.20 / 21.24
Grid Brightness Correction	24.30 / 18.66	0.85 / -	21.84 / 18.18
Running Average	26.71 / 25.67	21.12 / -	21.37 / 19.95
Frame Blending	27.27 / 24.61	23.30 / -	23.41 / 19.91
Frame Blending + Brightness Correction	25.53 / 23.61	22.53 / -	23.48 / 19.34
HSV-Blending	27.15 / 26.39	23.88 / -	22.81 / 21.05
Completed Cell	26.65 / 25.00	25.89 / -	20.28 / 20.67
Completed Cell + Brightness Correction	27.09 / 25.12	23.53 / -	21.67 / 20.76

Table 6.7: Render speed for the diverse image refinement approaches on the *SGS2* with *Android 2.3.5 Gingerbread* (1^{st} column), *LG-4xHD* with *Android 4.0.3 Ice Cream Sandwich* (2^{nd} column) and *SGS3* with *Android 4.0.3 Ice Cream Sandwich* (3^{rd} column) for resolutions of 2048x512 and 4096x1024 pixels. The maximum texture size of the LG-4xHD is 2048x2048 pixels.

that time. However, since the interest of this work concerning the render speed is limited to frame rate that is reached when mapping pixels, it is not part of this test.

Also a tolerance of 1-2 FPS has to be taken into account, due to the varying results of the tests.

Table 6.7 displays the render speed for the SGS2, the LG-4xHD and the SGS3 for lower and higher resolution panoramic images.

SGS2: For the lower resolution of 2048x512 pixels the tests for all image refinement approaches are above or close to 25 FPS and therefore run in real-time. The fastest mapping approach is the one without any refinements and the slowest approach is the grid brightness correction, which already indicates slightly more expensive calculation costs for determining the offset for the brightness correction. This effect is visible more significantly for the higher resolution panoramic image, in which the frame rate drops from 24.3 to 18.66 FPS. While the other tests can still be considered as real-time capable, the grid brightness correction falls a bit below that

speed limit.

LG-4xHD: With this mobile phone only tests for the lower (standard) resolution can be made, due to the maximum texture size of 2048x2048 pixels. All tests run in above 20 FPS, which can be considered as fluent, except the grid brightness correction. This specific test has a frame rate of under 1 FPS and is not applicable for practical use. The reason for that is the more expensive calculations in the fragment shader. This points out that the bottle neck for the *LG-4xHD* for this application is the GPU. Whereas the *SGS2* does the same test in real-time the *LG-4xHD* cannot achieve an acceptable speed. However the tests that also need some time for preparing the data for the shader, such as the completed cell approach, still run in real-time.

SGS3: Despite of the higher computational power of this mobile phone, the results cannot keep up with the *SGS2*. This is surprising, but the reason for that seems to be the different *Android* versions (*Ice Cream Sandwich* versus *Gingerbread*). The results themselves are constantly above 20 FPS for the standard resolution and around 18-20 FPS for the high resolution panoramic image. In case of the *SGS3*, no significant outliers regarding the render speed can be detected.

To get a better overview over all tests Figure 6.6 displays all results for each testing device and each refinement approach.

6.4 Interpretation of the Evaluation Results

To interpret the results described in the previous section all aspects, such as image quality, tracking accuracy and render speed, have to be considered.

Starting with the quality of the panoramic images from the perceptual point of view the frame-blending image refinement approach with a brightness correction achieves the most continuous and best results. The result images might seem a bit blurry in some regions, but seams are significantly reduced and other image artifacts generated in the CPU-mapping are removed. The approach is dependent of the movement of the camera, which means it creates different results, concerning the image quality, by moving the camera differently towards or away from light sources. This is the case for all approaches that map the whole camera image for each frame. The camera movement depending from the light source also affects the image refinement approaches that only map new pixels for each frame, but a seam of

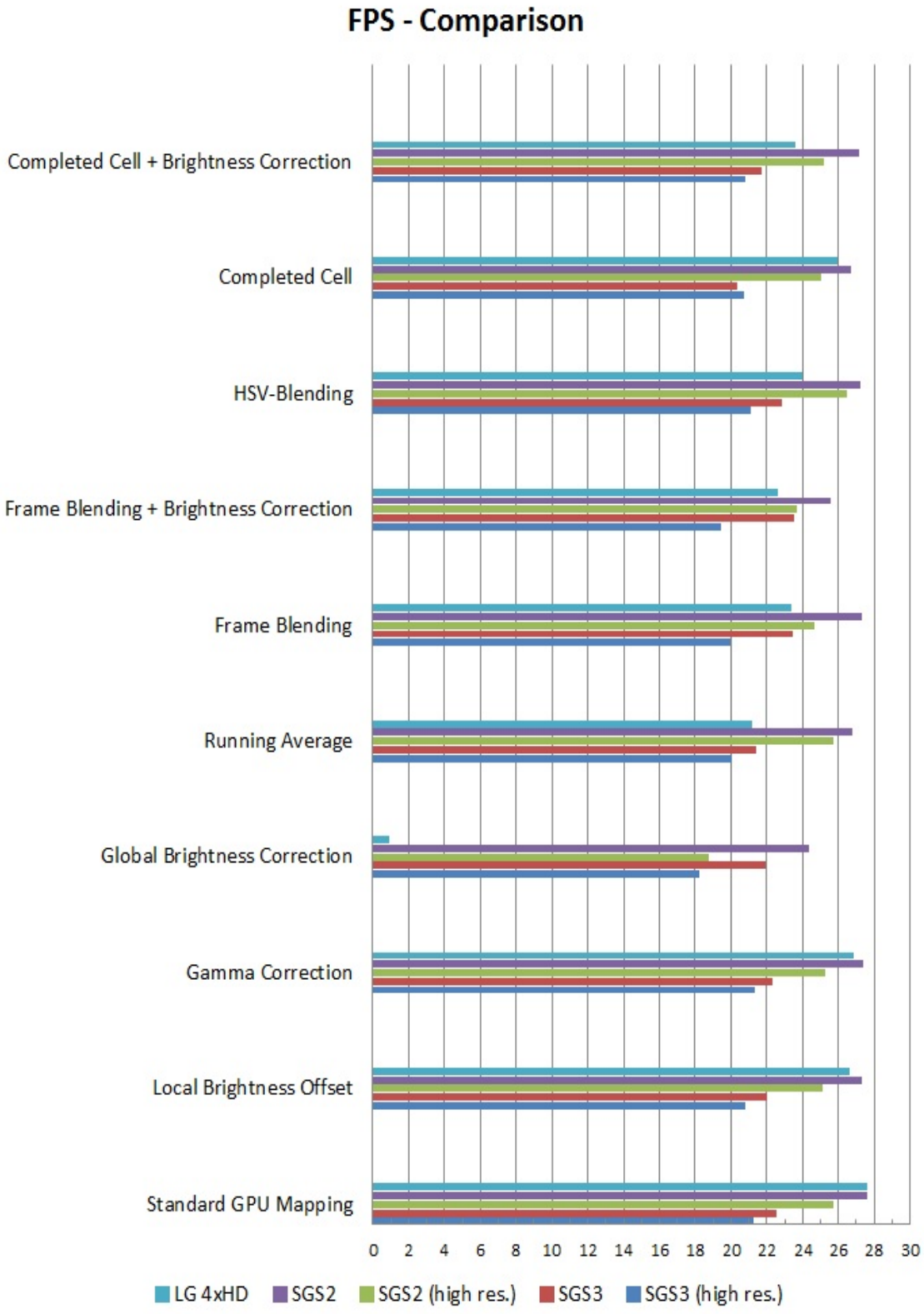


Figure 6.6: Overview of the render speed results for the diverse image refinement approaches

brightness differences will be visible in any way. Since image sequences created from videos are taken as input for the tests, the blending approaches could not correct all visible seams in the tests. In practical use the user can always remap areas where seams appear. The seams can be discovered in the panoramic image preview and by moving the camera over the scene where a seam has been located, the seam can possibly be removed.

For tracking the camera image in the panoramic map using the FAST corner detection algorithm, the sharpness of corners and edges are of most importance. Therefore image refinement approaches that only map new pixels achieve better results than the ones mapping the whole camera image. Strong differences in brightness however can force the tracker to loose its orientation and it needs to relocate the orientation. This costs additional computation time and is disturbing in practical use. Since all approaches achieve acceptable tracking results, the image quality and render speed are used to decide which image refinement approach is to prefer. In general approaches that update only pixels that have not been mapped before achieve a better tracking score than in the CPU-mapping.

Concerning the render speed for the standard resolution of 2048x512 pixels, all image refinement approaches can be used except for the grid brightness correction, since it does not work fast for the *LG-4xHD*. All the other approaches run fluently with a frame rate higher than 20 FPS. Similar to lower resolutions, when rendering a higher resolution panoramic image (4096x1024 pixels) the frame rate is about 20 FPS or higher for all approaches except the grid brightness correction. Therefore, when excluding the grid brightness correction approach, all refinement approaches can be chosen with regard to the rendering speed.

The GPU of the *SGS2* and *SGS3* (*Mali-400MP*) seems to operate faster than the one of the *LG-4xHD* (*Tegra 3*) for this application. The grid brightness correction needs additional calculation for preparing data for the fragment shader and in the fragment shader as well. Skipping the calculation on the CPU side still results in similar frame rates than with it.

The conclusion of the evaluation is that most of the approaches achieve a real-time frame rate for the standard resolution and still show a frame rate above 20 FPS for the high resolution. The tracking works best using image refinement approaches that only map new pixels per frame, but is still acceptable for the others. The image quality is optimized using the frame-blending approach with the feature point brightness offset correction, which is the most preferable approach regarding the results.

6.5 Implementation Issues

An implementation of the GPU-based mapping approach reveals several issues that have to be taken care of.

Rendering Framework: The most crucial issue emerges using the Open Scene Graph (OSG) framework. Despite a structured and convenient programming environment, the maximum render speed cannot exceed 17 FPS using the *Samsung Galaxy S II* (SGS2) as a testing device. Even optimizations of the shader's viewport by passing it through a scissor test did not achieve real-time frame rates. An implementation in plain OpenGL ES 2.0 delivers a desired render speed of 25 FPS and above. We speculate that the overhead in scene graph traversals within OSG is the reason for the suboptimal behavior.

Texture Size: Increasing the size of the panoramic image requires an enlargement of the texture resolution. Some mobile phones do not support texture sizes beyond 2048x2048 pixels, which is a hardware dependent limit varying for different kinds of mobile phones. Whereas the *SGS2* has a maximum texture size of 4096x4096 pixels, the maximum texture size of the *LG Optimus 4x HD* is only 2048x2048 pixels. An implementation of the panoramic mapping approach has to take the maximum texture size supported into account. This can either be done by enforcing a limit on the maximum size of the panoramic image, or by adapting the implementation to use multiple textures for example.

Dithering: A device specific problem for the Samsung Galaxy series concerns the default behavior of the GPU deployed. The graphics chip *Mali 400* automatically sets dithering in Frame Buffer Objects (FBO) to enabled. Creating panoramic images not disabling this setting result in strange symmetirc patterns that get amplified through blending the camera images. The difference of the output images can be seen in Figure 6.7(a) and 6.7(b).

(a) dithering enabled (b) dithering disabled

Figure 6.7: Snippet of a panoramic image created with and without dithering

Chapter 7

Concluding Remarks and Future Work

In this work, a GPU-based approach for mapping panoramic images is proposed. Acquiring the orientation of the camera in a cylindrical environment using a tracker, a rotation matrix can be generated. This matrix allows to determine the mapping location of the camera image in the environment in cylinder space. Determining if a pixel of the panoramic image is in the estimated area of the camera image can be implemented very efficiently on the GPU using shader, since it is heavily parallelizable.

A problem for existing real-time systems on mobile phones is handling lighting differences, since there is no programmable control of the exposure time. As demonstrated in this work, a shader implementation automatically allows to blend the current camera frame with already mapped pixels and therefore generates better results in terms of image quality. A comparison of several methods for reducing brightness seams reveals that the most promising approach is to blend the camera image from each side to the panoramic map along the extents of the currently mapped frame. Most blending methods developed and investigated during this thesis operate with more than 25 FPS on recent mobile phones. Although it is not possible to remove all noticeable brightness differences completely, however, the results presented demonstrate a significantly improvement in image quality still maintaining real-time performance.

Blending images often results in a bit of blurriness. This is expected to have a negative influence on the performance of a tracker working in the panoramic space. To visualize the performance degradation of such a tracker using the method of [Wagner10], the found feature points per image and the number of matches are compared between panoramic images mapped on the CPU and on the GPU. As it is shown, in general the GPU-mapped panoramic images achieved a higher score. In terms of tracking performance, however, approaches without any kind of pixel-blending performed a bit better than those that blend images. Nevertheless the images generated employing blending methods can compete with pure CPU-based map-

ping approaches and deliver a better image quality.

A further advantage of GPU-based mapping is the possibility of creating higher resolution panoramic images without suffering from a significant deterioration in render speed. As it is shown, on recent mobile phones a frame rate between 20 and 26 FPS is achieved for rendering textures with the size of 4096x1024 pixels.

In this work, additional functionality is applied that improves the usability of an application, while leveraging the benefits of a GPU-based implementation, such as the proposed wiping function. Since taking a panoramic image is a prolonged process, parts of the scene can easily be covered by moving objects or become unusable due to persons that change positions. To cope with that problem the wiping function enables the user to delete parts of the panoramic image online and remap the cleared areas.

7.1 Future Work

Enhancements that can be done on recent devices could include changing the tracking method to remove CPU-mapping, allowing even bigger panoramic images by splitting up the texture map into several smaller ones or adding additional functionality.

Remove CPU Mapping: The tracking method used in this work is based on [Wagner10]. This method automatically maps on the CPU and uses the generated panoramic image for tracking. Simply removing the CPU-based mapping process is not possible, since reading back the updated panoramic image from the GPU-memory to the CPU-memory is a very time consuming operation and cannot be used for real-time mapping and tracking. For instance a tracking approach operating with key-frames does not require to read the panoramic image back from the GPU-memory and therefore reduces computational redundancies.

Split Textures for High Resolution Panoramic Images: Proceeding from a naive implementation, for some devices adaptive shader have to be realized. As discussed in Section 5.3 the resolution can be increased without a noticeable loss of render speed. The current implementation. simply enlarges the texture size of the panoramic map, which works fine for some devices. Due to limited texture sizes the maximum resolution depends on the hardware. To create even larger panoramic images, one could split the main panoramic texture into several parts

and adapt the calculation for each texture. Due to varying precisions of floating point values, fixed point calculation can be required for an accurate mapping process.

Additional Functionality: Larger dynamic areas can automatically be recognized by a background model. To determine whether an area is dynamic or not the current camera image can be subtracted from the panoramic image, resulting in a black image region. Areas that exceed a given threshold can be marked as dynamic. These regions of differences (ROD) can be discarded for mapping or intentionally marked for remapping on the panoramic image. This procedure can be processed very efficiently using a shader implementation, since the respective coordinates of both, the panoramic map and the camera image are known during the mapping process.

Even if creating panoramic images is a widely discussed topic, new hard- and software developments will allow further improvements in the quality of images. Especially using graphics processors with their programmable pipeline and the steadily increasing computing power on mobile phones will play major roles in future image processing on handheld devices.

List of Figures

List of Tables

Bibliography

[Adams08] A. Adams, N. Gelf, and K. Pulli. *Viewfinder alignment. Computer Graphics Forum (Proc. Eurographics*, pp. 597–606, **2008**.

[Azuma97] R. T. Azuma. *A survey of augmented reality. Presence: Teleoperators and Virtual Environments*, volume 6(4):pp. 355–385, **1997**.

[Baudisch05] P. Baudisch, D. Tan, D. Steedly, E. Rudolph, M. Uyttendaele, C. Pal, and R. Szeliski. *Panoramic viewfinder: providing a real-time preview to help users avoid flaws in panoramic pictures.* In *In Proceedings of the 17th Australia conference on Computer-Human Interaction: Citizens Online: Considerations for Today and the Future*, pp. 1–10. **2005**. ISBN 1-59593-222-4.

[Brown03] M. Brown and D.G. Lowe. *Recognising Panoramas.* In *IEEE International Conference on Computer Vision*, volume 2, pp. 1218–1225. **2003**.

[ITU-R90] Recommendation ITU-R BT.709. *Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange*, **1990**. Formerly CCIR Rec. 709, Geneva, Switcherland.

[Davison03] A. Davison. *Real-Time Simultaneous Localisation and Mapping with a Single Camera.* In *ICCV '03 Proceedings of the Ninth IEEE International Conference on Computer Vision*, pp. 1403–1410. **2003**.

[Degendorfer10] C. Degendorfer. *Mobile Augmented Reality Campus Guide.* Master's thesis, Graz University of Technology, **2010**.

[DiVerdi08] S. DiVerdi, J. Wither, and J. Höllerer. *Envisor: Online Environment Map Construction for Mixed Reality.* In *In Proc. IEEE VR 2008 (10th Intl Conference on Virtual Reality.* **2008**.

[Farbman09] Z. Farbman, G. Hoffer, Y. Lipman, D. Cohen-Or, and D. Lischinski. *Coordinates for instant image cloning. ACM Transactions on Graphics*, volume 28(12):pp. 1–9, **2009**.

[ICG09] Institute for Computer Graphics and Vision. *Handheld Augmented Reality*, **2009**. Available at: `http://studierstube.icg.tu-graz.ac.at/handheldar/index.php`, last visited: 2013.01.14.

[Klein07] G. Klein and D. Murray. *Parallel Tracking and Mapping for Small AR Workspaces*. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, pp. 1–10. **2007**.

[Lovegrove10] S. Lovegrove and A. Davison. *Real-time spherical mosaicing using whole image alignment*. In *The 11th European Conference on Computer Vision (ECCV 2010)*, pp. 73–86. **2010**.

[Lowe04] D.G. Lowe. *Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision*, volume 60(2):pp. 91–110, **11 2004**.

[López09] M.B. López, J. Hannuksela, O. Silvén, and M. Vehviläinen. *Graphics hardware accelerated panorama builder for mobile phones*. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 72560D–72560D–9. **2009**.

[Milgram94] P. Milgram and F. Kishino. *A Taxonomy of Mixed Reality Visual Displays. IEICE Transactions Information Systems*, volume E77-D(12):pp. 1321–1329, **1994**.

[Montiel06] J.M.M. Montiel. *A visual compass based on slam*. In *Proceedings 2006 IEEE International Conference on Robotics and Automation*, pp. 1917–1922. **2006**.

[Munshi10] A. Munshi and J. Leech. *OpenGL ES Common Profile Specification*, **2010**. Version 2.0.25 (Full Specification). Available at: `http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`, last visited: 2013.01.14.

[OSG07] OpenSceneGraph. *Introduction*. Available at: `http://www.openscenegraph.org/projects/osg/wiki/About/Introduction`, last visited: 2013.01.14.

[Plataniotis00] K. N. Plataniotis and A. N. Venetsanopoulos. *Color image processing and applications*. Springer-Verlag, New York, first edition, **2000**. ISBN 978-3-540-66953-1.

[Pulli10] K. Pulli, M. Tico, and Y. Xiong. *Mobile Panoramic Imaging System.* In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pp. 108–115. **2010**.

[Rosten06] E. Rosten and T. Drummond. *Machine learning for high-speed corner detection.* In *European Conference on Computer Vision*, volume 1, pp. 430–443. **2006**.

[Schall09] G. Schall, D. Wagner, G. Reitmayr, E. Taichmann, M. Wieser, D. Schmalstieg, and B. Hofmann-Wellenhof. *Global Pose Estimation Using Multi-Sensor Fusion for Outdoor Augmented Reality.* In *Virtual Reality Conference (VR), 2010 IEEE*, pp. 153–162. **2009**.

[Smith78] A.R. Smith. *Color gamut transform pairs.* In *SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, volume 12, pp. 12–19. **1978**.

[Steedly05] D. Steedly, C. Pal, and R. Szeliski. *Efficiently Registering Video into Panoramic Mosaics.* In *Tenth IEEE International Conference on Computer Vision, 2005 ICCV*, volume 1, pp. 1300–1307. **2005**.

[Szeliski06] R. Szeliski. *Image Alignment and Stitching: A Tutorial. Foundations and Trends in Computer Graphics and Vision*, volume 2:pp. 1–104, **2006**.

[Szeliski97] R. Szeliski and H. Y. Shum. *Creating Full View Panoramic Image Mosaics and Environment Maps.* In *24th Annual Conference on Computer Graphics - SIGGRAPH, 1997*, pp. 251–258. **1997**.

[Tian02] G.Y. Tian, D. Gledhill, D. Taylor, and D. Clarke. *Colour correction for panoramic imaging.* In *Proceedings Sixth International Conference on Information Visualisation*, pp. 483–488. **2002**.

[Uyttendaele01] M. Uyttendaele, A. Eden, and R. Szeliski. *Eliminating Ghosting and Exposure Artifacts in Image Mosaics.* In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 509–516. **2001**.

[Wagner10] D. Wagner, A. Mulloni, T. Langlotz, and D. Schmalstieg. *Real-Time Panoramic Mapping and Tracking on Mobile Phones.* In *Virtual Reality Conference (VR), 2010 IEEE*, pp. 211–218. **2010**. ISSN 1087-8270.

[Wagner09a] D. Wagner and D. Schmalstieg. *Making Augmented Reality Practical on Mobile Phones, Part 1*. In *Computer Graphics and Applications, IEEE 2009*, volume 29, pp. 12–15. **2009**.

[Wagner09b] D. Wagner and D. Schmalstieg. *Making Augmented Reality Practical on Mobile Phones, Part 2*. In *Computer Graphics and Applications, IEEE 2009*, volume 29, pp. 6–9. **2009**.

[Wang10] R. Wang and X. Qian. *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, Birmingham, **2010**. ISBN 978-1-849512-82-4.

[Xiong09b] Y. Xiong and K. Pulli. *Gradient Domain Image Blending and Implementation on Mobile Devices*. In *Mobicase 2009*. **2009**.

[Xiong10] Y. Xiong and K. Pulli. *Fast Panorama Stitching for High-Quality Panoramic Images on Mobile Phones*. In *The 7th IEEE conference on Consumer communications and networking conference*, pp. 537–541. **2010**.

[Xiong09a] Y. Xiong, X. Wang, M. Tico, C.K. Liang, and K. Pulli. *Panoramic imaging system for mobile devices*. In *Poster at the 36th international conference and exhibition on Computer graphics and interactive techniques (SIGGRAPH 2009)*. **2008**.