Panoramic Mapping on Mobile Phone GPUs

Georg Reinisch*
Supervised by: Clemens Arth†

Institute of Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Creating panoramic images in real-time is an expensive operation for mobile devices. Depending on the size of the camera image the mapping of individual pixels into the panoramic image is one of the most time consuming parts. This part is the main focus in this paper and will be discussed in detail. To speed things up and to allow 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 on the panoramic image allows OpenGL 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 fluently on current generation devices.

Keywords: Mobile Phone, GPU, OpenGL, Panoramic Image, Augmented Reality

1 Introduction

In the literature, different kinds of techniques are proposed for the purpose of generating panoramic images. For the creation of such panoramic images out of several regular photographs, images that are composed and aligned using stitching or image mosaic algorithms are required [13][14]. Most of the recent approaches that generate panoramic images do this in an offline process [12][14].

For AR purposes, Wagner *et 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 [15]. 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.

Since the existing real-time panoramic mapper and tracker solely works on the CPU, it can only handle a small image resolution. 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 Wagner's approach is that it eliminates the chance of blending pixels to cover seams generated by brightness differences.

In this work we complement the CPU-based rendering approach by Wagner *et al*. with a GPU-based implementation to transfer computational costs from the CPU to the GPU. 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 this work, methods are discussed for reducing or eliminating 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 that allows the user to remove unwanted areas of the panoramic image and remap them again.

2 Related Work

For aligning images several approaches exist that are suitable for different types of cases. A tracking method described by Lowe [9] that searches for scale invariant key points (SIFT), is used in several offline approaches. Most existing approaches for panoramic imaging or creating image mosaics work offline [3, 12, 14].

Adam *et al*. in [1] discuss a method in which the successive images of a camera's view finder are aligned online and in real-time. However this method does not permit to create closed 360 degree images and to track the 3D motion of the phone. In [16] the viewfinder algorithm is used for tracking the camera motion to create high resolution panoramic images. The whole approach itself does not run in real-time and requires offline processing.

Baudisch *et al*. in [2] generate a low resolution real-time preview while shooting the panoramic image. Similar to the approach described in this paper, the preview is used to avoid missing to capture relevant areas or that relevant areas disappear when cropping the panoramic image to its

*georg.reinisch@student.tugraz.at
†arth@icg.tugraz.at

rectangular shape.

A similar real-time tracking method as in [15] is described by DiVerdi *et al*. in [5]. The system called *Envisor* is capable of creating environment maps online. For refining optical flow measurements DiVerdi *et al*. use landmark tracking, which requires extensive GPU resources for real-time processing and does not run on mobile phones.

GPU-acceleration: 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 [7]. Pulli *et al*. warp their images via spherical mapping calculated on the GPU [10]. Since López *et al*. and Pulli *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.

An approach that creates spherical image mosaics in real-time using graphics processors for faster computation is discussed by Lovegrove *et al*. [8]. However this approach does not run on mobile phones in real-time, since the computational power of handheld devices is very limited.

Image Refinement: Removing the seams of panoramic images that occur if two images with different illumination are stitched together is a widely discussed topic. 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*. [8] sums up the pixels' color values and divides them through the number of times the pixel has been mapped.

Pulli *et al*. in [10] use a fast image cloning approach for transition smoothing based on [6], which runs in real-time for desktop-GPUs and delivers seamless results, but cannot be computed online on mobile phones.

As an extension to the approach of Wagner *et al*. [15], Degendorfer [4] implemented a brightness correction method to enhance the image quality with an extended dynamic range. The strength of the seams is reduced, but they are not eliminated completely.

In summary, all approaches mentioned are either not running in real-time on mobile phones or cannot eliminate artifacts such as ghosting or brightness seams completely.

3 Panoramic Mapping and Tracking

The tracking approach used for this work was taken from Wagner *et al*. [15]. 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. Wagner's 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.

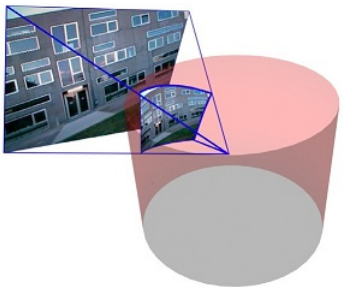

Figure 1: Projection of the camera image on the cylindric map. [15]

In the following, the tracking and mapping approach by Wagner *et al*. is described in more detail.

Panoramic Tracking: To estimate the location of the current camera image for the mapping process, the new image has to be tracked accurately. Therefore the FAST corner detector [11] for feature point extraction is used, ranking the feature points found by strength. To get a valid tracking result, the number of the corner points must exceed a given threshold.

For the tracking process 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. On success, the rotation matrix acquired is used in the mapping process to project the current camera image onto the map.

Panoramic Mapping: As a mapping surface, a cylinder is chosen. The panoramic map is split up into a regular grid of 32x8 cells, 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.

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 image. 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.

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), the computational power is reduced significantly.

4 GPU Shader Implementation

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 parallelized and hence they are ideal to calculate on the GPU. Since the mapping is completely independent for each individual pixel, the idea of this paper is to compute this part in a shader-program 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.

For blending pixels, information about the current panoramic image is required. Therefore a render-to-texture approach using two framebuffers has been chosen and a common method also known as "ping-pong technique" has been applied.

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 information 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.

Shader Data Preparation: 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 angular resolution, the model for the parametrization of the surface needs to be known. As suggested in [15] we chose a cylindrical model for the mapping procedure. 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 can be arbitrarily chosen, but in our case a 4 by 1 ratio is used. The height h of the cylinder is therefore set to $C/4.0$. 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:

$$a = \frac{C}{W} \qquad b = \frac{h}{H} \tag{1}$$

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. During the render cycles the rotation matrix will be calculated externally in the tracking process. 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:

$$\vec{r} = \mathbf{R}^{\mathbf{T}} \, \vec{d} \tag{2}$$

For the calculation of the projection matrix the calibration matrix $\boldsymbol{K}$ (generated in an initialization step), the rotation matrix $\boldsymbol{R}$ (calculated in the tracking process) and the camera location $\vec{t}$ is required. Since the camera is located in the center of the cylinder ($\vec{t}$*(0, 0, 0)*), calculating $\boldsymbol{P}$ can be simplified by multiplying $\boldsymbol{K}$ by $\boldsymbol{R}$.

Shader Calculations: After preparing this information the data is sent to the fragment shader via uniforms. The coordinates of the input/output texture u and v (framebuffer-switching) 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:

$$c_x = \sin(u\,a) \qquad c_y = v\,b \qquad c_z = \cos(u\,a) \tag{3}$$

a and b are the angle resolutions as given in Equation 1.

When projecting a camera image on the cylinder it is projected twice (once on the front-side and once on the back-side that is flipped). To avoid mapping the image twice, a check whether the cylinder coordinates are in the front of the camera or in the back is performed. The check eliminates the back side of the cylinder.

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.

$$i_x = \mathbf{P_{0,0}}\, c_x + \mathbf{P_{1,0}}\, c_y + \mathbf{P_{2,0}}\, c_z \quad (4)$$

$$i_y = \mathbf{P_{0,1}}\, c_x + \mathbf{P_{1,1}}\, c_y + \mathbf{P_{2,0}}\, c_z \quad (5)$$

$$i_z = \mathbf{P_{0,2}}\, c_x + \mathbf{P_{1,2}}\, c_y + \mathbf{P_{2,0}}\, c_z \quad (6)$$

To get the image point the homogenous coordinates are converted to image coordinates.

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 about 1 Mio. fragments every operation done in the shader will be executed over one million times. Even when discarding the shader program as soon as 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.

4.1 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. This reduces the maximal number of shader runs from about 1 Mio. 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 mapping those that were not mapped before. Assuming a panoramic image is tracked in real-time the frame

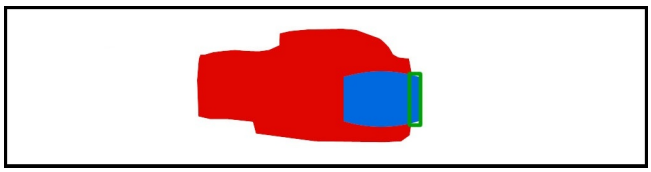

Figure 2: Red: mapped area; blue: current frame; green: small update region that is cut by the scissor test and passed to the shader. The additional optimization approach saves comptation costs.

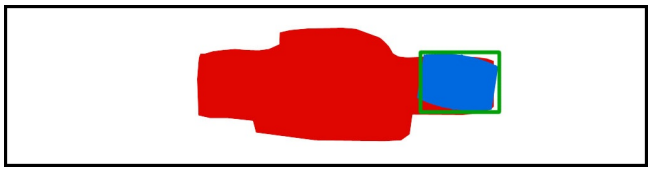

Figure 3: Red: mapped area; blue: current frame; green: big update region that is cut by the scissor test and passed to the shader. The additional optimization approach does not save a lot of computation costs.

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 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 2).

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 3.

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 calculating a bounding-box around the camera frame.

4.2 Wiping

Using a mapping approach running on the GPU allows us to add new features, such as the possibility to wipe out areas in the panoramic images in real-time. Panoramic images happen to contain unwanted areas like persons or

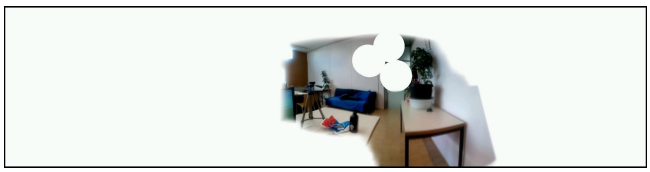

Figure 4: Circular white spots have been cleared while taking the panoramic image

Figure 5: Sharp edges in homogenous areas due to diverging exposure time

Figure 6: Brightness offset correction calculated from feature points [4]

cars that cover an essential part of the scene. To remove these unwanted spots the panoramic image can be edited in real-time by wiping over the panoramic image preview displayed on the mobile phone's screen. For example, by specifying an area in a preview image of the 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 wipe 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 coordinate $\vec{t}$ and the marked wiping coordinates $\vec{w}$.

$$(\vec{t} - \vec{w}) \cdot (\vec{t} - \vec{w}) < (N^2) \tag{7}$$

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.

4.3 Image Refinement

A significant problem while taking panoramic images in real-time is the changing exposure time of the camera, which can usually not be fixed on current mobile phones. When moving the camera towards a light source the exposure is reduced, which significantly darkens the input image. Moving the camera away from the light source again brightens 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.

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 become an active part of the real-time capturing of a panorama for certain approaches.

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 by Degendorfer [4] is revised. Degendorfer calculates the brightness offset for the feature points found by the tracker (see Figure 6). 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.

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 were investigated, however, a frame-based blending approach turned out to achieve the best continuously 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 7. Pixels at the image border (outer frame) are taken from the panoramic map. A linear blending operation is used in the

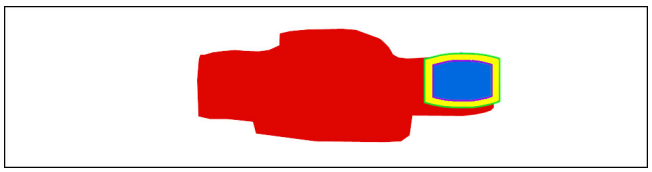

Figure 7: Linearly blending the camera image with the panoramic image in the frame area (yellow) between the outer (green) and the inner (purple) blending frame.

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.

The following pseudo code represents the blending algorithm, where x and y are the camera image coordinates, *frameWidth* is the width of the blending frame, *camColor* and *panoColor* are the colors of the respective pixels of the camera and panoramic image and *alphaFactor* is the calculated blending factor:

Algorithm 1 Frame Blending

```
Require: fragment in camera frame
  if fragment in blending frame) then
    if alreadyMapped == TRUE) then
      minX = x > frameWidth ? camWidth - x : x;
      minY = y > frameWidth ? camHeight - y : y;
      alphaFactor = minX < minY ? minX/frameWidth :
      minY/frameWidth;
      newColor.r = camColor.r*alphaFactor +
      panoColor.r*(1.00-alphaFactor);
      newColor.g = camColor.g*alphaFactor +
      panoColor.g*(1.00-alphaFactor);
      newColor.b = camColor.b*alphaFactor +
      panoColor.b*(1.00-alphaFactor);
    else
      color = camColor;
    end if
  else
    color = camColor;
  end if
```

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. Naturally, the blending operations can be combined with the brightness offset correction.

4.4 Large panoramic images

Mapping a panoramic image on a CPU in real-time is possible for medium-size panoramic images only. 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 negligible loss in 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.

5 Experimental Results

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 quality is determined from a perceptual point of view for achieving continuous results without seams and artifacts. 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 experimental results highlight which image refinement algorithm performs best in terms of quality, robustness and speed. The results are also compared with the CPU-side mapping approach by Wagner *et al*. [15].

5.1 Panoramic image refinement

When taking a panoramic image, 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 exacerbates this test scenario. However distinctive shadow structures enable the tracker to find corresponding points on otherwise homogenous regions.

Figure 8(a) is the reference image created by the original application ([15]). 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 as described by [4] 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 8(b).

In Figure 8(c) the seams as well as general differences in brightness are smoothed by the blending approach com-

(a) CPU-mapped image created by the application of Wagner *et al*. [15]

(b) GPU-mapped image with brightness offset correction

(c) GPU-mapped image with blending and brightness offset correction

Figure 8: Panoramic images of a sunny scene

bined with the brightness offset correction. As a result of the smoothing, 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 blending approaches are dependent of the movement of the camera, which means that they create different results, concerning the image quality, by moving the camera differently towards or away from light sources. Since the test result was generated from an image sequence of a taken video, still visible artifacts could not be removed. In practical use the user would have recognized the gray area and its seams, due to the panoramic preview and remapped it by moving the camera from a different direction over that region.

5.2 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.

Table 1 shows the 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.

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
Frame Blending	73.95	1028.54
Frame Blending + Brightness Correction	78.03	1030.62

Table 1: Average of found key and matching points for the respective refinement approach

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.

5.3 Render Speed

The speed tests discussed in this section measure the averagely rendered frames per second for each image refine-

Approach	FPS: SGS2 (low/high)	FPS: LG (low/high)	FPS: SGS3 (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
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

Table 2: Render speed for the diverse image refinement approaches on the *SGS2* (1^{st} column), *LG* (2^{nd} column) and *SGS3* (3^{rd} column) for resolutions of 2048x512 and 4096x1024 pixels. The maximum texture size of the LG is 2048x2048 pixels.

ment 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:

- Samsung Galaxy S II (SGS2): 1.2GHz dual core; Mali-400MP; Android 2.3.5
- LG Optimus 4x HD (LG): 1.5GHz quad core; Nvidia Tegra 3; Android 4.0.3
- Samsung Galaxy S III (SGS3): 1.4GHz quad core; Mali-400MP; Android 4.0.3

Table 2 displays the render speed for the SGS2, the LG and the SGS3 for lower and higher resolution panoramic images.

Concerning the render speed for the standard resolution of 2048x512 pixels, all image refinement 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. Despite of the higher computational power of the *SGS3*, 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*).

6 Conclusions

In this paper, a GPU-based approach for mapping panoramic images is proposed. Investigating several methods that address the auto-exposure of cameras, artifacts and brightness seams can be eliminated or strongly reduced. The mapping process implemented on the GPU works very efficiently using shader programs, since pixel-mapping is heavily parallelizable. This allows larger panoramic images to be generated in real-time and additional functionality, such as wiping is added.

The proposed GPU-based mapping approach still requires the mapping on the CPU-side for tracking, which can be replaced by key-frame tracking as future work.

References

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

[2] 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 *Australian Conf. on Computer-Human Interaction*, pages 1–10, 2005.

[3] M. Brown and D.G. Lowe. Recognising Panoramas. In *ICCV*, volume 2, pages 1218–1225, 2003.

[4] C. Degendorfer. Mobile augmented reality campus guide. Master's thesis, Graz University of Technology, 2010.

[5] S. DiVerdi, J. Wither, and J. Höllerer. Envisor: Online Environment Map Construction for Mixed Reality. In *VR*, 2008.

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

[7] 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*, pages 72560D–72560D–9, 2009.

[8] S. Lovegrove and A. Davison. Real-time spherical mosaicing using whole image alignment. In *ECCV*, pages 73–86, 2010.

[9] D.G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *IJCV*, 60(2):91–110, 11 2004.

[10] K. Pulli, M. Tico, and Y. Xiong. Mobile Panoramic Imaging System. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 108–115, 2010.

[11] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV*, volume 1, pages 430–443, 2006.

[12] D. Steedly, C. Pal, and R. Szeliski. Efficiently Registering Video into Panoramic Mosaics. In *ICCV*, volume 1, pages 1300–1307, 2005.

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

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

[15] D. Wagner, A. Mulloni, T. Langlotz, and D. Schmalstieg. Real-Time Panoramic Mapping and Tracking on Mobile Phones. In *VR*, pages 211–218, 2010.

[16] Y. Xiong, X. Wang, M. Tico, C.K. Liang, and K. Pulli. Panoramic imaging system for mobile devices. In *Poster at Int. Conf. and Exhib. on Computer graphics and interactive techniques (SIGGRAPH 2009)*, 2008.