Real-time Hiding of Physical Objects in Augmented Reality

Niclas Scheuing

Master Thesis
August 2018

Dr. Stéphane Magnenat
Henry Raymond
Prof. Dr. Bob Sumner
Abstract

Image inpainting, the process of reconstructing missing pixels, has been intensely studied for almost two decades and many solutions to the problem have been proposed. Inpainting of videos has also been discussed, mostly in the context of post-processing and there are very few publications on real-time video inpainting. The widespread availability of mobile devices opens a new range of applications of image and video inpainting. Especially in the context of augmented reality, a solution to the video inpainting problem that runs in real-time on mobile devices is most useful for a wide range of applications.

We therefore propose a video inpainting algorithm and present a GPU implementation thereof. We embed this implementation into an augmented reality application that removes a tracked object in real-time and runs on iOS, Android and Windows. Our algorithm is based on the PixMix approach introduced by Herling and Broll, which is inspired by patch-matching algorithms, but matches individual pixels instead of entire patches. We use a gradient descent optimization to minimize a cost function composed of a term enforcing color consistency and one preserving texture patches.

In this thesis we will show that real-time video inpainting on mobile devices is feasible without any prior knowledge on the setting, provided that the computational power of the GPU is exploited. We demonstrate how this can be used in combination with image tracking to remove a known physical object.
Contents

List of Figures vii

List of Tables ix

1. Introduction 1
   1.1. Problem Description ............................................. 1
   1.2. Goal and Final Product ........................................... 2
   1.3. Constraints .......................................................... 2
   1.4. Our Approach ....................................................... 3

2. Related Work 5
   2.1. Image Inpainting ................................................... 5
       2.1.1. Diffusion-based Algorithms ................................. 6
       2.1.2. Patch-based Algorithms ..................................... 6
       2.1.3. Hybrid Methods ................................................. 7
   2.2. Video Inpainting ................................................... 7
   2.3. Inpainting Quality Assessment .................................... 8
   2.4. Approach and Rationale .......................................... 8
       2.4.1. Modeled, Learned or Synthesized Background ............... 9
       2.4.2. Diffusion or Patches .......................................... 11
   2.5. Inpainting problem as Optimization .............................. 13
   2.6. The PixMix Algorithm ............................................. 13
       2.6.1. Cost function .................................................. 13
       2.6.2. Image inpainting .............................................. 14
       2.6.3. Video inpainting .............................................. 15
# Contents

3. Method 17
   3.1. Our Approach ................................. 17
      3.1.1. Optimization ............................ 17
      3.1.2. Other Modifications .................... 20
   3.2. Algorithm .................................. 22
      3.2.1. Initialization ............................ 22
      3.2.2. Optimization ............................ 22
      3.2.3. Mapping Propagation to Next Mipmap Level 24
      3.2.4. Mapping Propagation to Next Frame ..... 25

4. Implementation 27
   4.1. GPU Implementation .......................... 27
      4.1.1. Coding for GPUs ......................... 28
      4.1.2. Image Tracking ............................ 29
      4.1.3. Scene Setup ............................... 29
      4.1.4. Mapping Function as Texture .......... 30
      4.1.5. Initialization .............................. 31
      4.1.6. Neighborhoods ............................. 31
      4.1.7. Optimization ............................... 32
      4.1.8. Mapping Propagation ...................... 32
      4.1.9. Mapping Texture Access .................. 33
   4.2. Python Prototype ............................ 34

5. Results 37
   5.1. Result Overview .............................. 37
   5.2. Method ...................................... 38
   5.3. Parameters .................................. 38
   5.4. Performance .................................. 39
   5.5. \( \alpha \)-Weight and Interpolation ............ 42
   5.6. \( \tau \)-Clamping of Spatial Gradient ............ 46
   5.7. Color Cost Normalization ...................... 47
   5.8. Mapping Back into the Hole .................. 48
   5.9. Spatial Cost Neighborhood .................... 48
   5.10. Undersampling and Rotation .................. 49
   5.11. Shadows ..................................... 49
   5.12. Rotation dependency .......................... 50

6. Conclusion 57

7. Future Work 59

A. Sample Images 61
   A.1. Own Images ................................. 61
      A.1.1. building .................................. 61
      A.1.2. whiteboard ................................ 62
      A.1.3. bunny .................................... 62
      A.1.4. mill ...................................... 63
List of Figures

2.1. Comparison of different object removal techniques. . . . . . . . . . . . . . . . . 12
3.1. Spatial cost term, color cost term and total cost. . . . . . . . . . . . . . . . . . 20
4.1. Schematic of the Unity scene. . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2. The neighborhoods used. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.1. Results produced using our algorithm. . . . . . . . . . . . . . . . . . . . . . . 38
5.2. Interpolation artifacts. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.3. Comparison of different values for $\tau$. . . . . . . . . . . . . . . . . . . . . 51
5.4. Comparison of different color cost normalizations. . . . . . . . . . . . . . . . 52
5.5. Comparison of three approaches to avoid mapping pixels into the target area. . 52
5.6. Comparison of three approaches to avoid mapping pixels into the target area. . 53
5.7. Comparison of different neighborhoods used in the spatial cost term . . . . . . . 54
5.8. Initialization with undersampling and without rotation invariance. . . . . . . . . 55
5.9. Shadow remaining after removing the physical object. . . . . . . . . . . . . . . . 55
5.10. Results depending on the image orientation . . . . . . . . . . . . . . . . . . . . 56
A.1. The image we call building and its three masks. . . . . . . . . . . . . . . . . . 61
A.2. The whiteboard image and its mask. . . . . . . . . . . . . . . . . . . . . . . . 62
A.3. The bunny image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.4. The mill image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.5. The zucchini image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.6. The tea cup image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.7. The whiteboard image and its mask. . . . . . . . . . . . . . . . . . . . . . . . 64
A.8. The lausanne1 image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . 65
A.9. The lausanne2 image and its mask. . . . . . . . . . . . . . . . . . . . . . . . . 65
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Default parameters.</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview on time spent on individual steps of the algorithm.</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Time measurements per mipmap level.</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance evaluation on a Nvidia Shield tablet.</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Parameters for Section 5.4.</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Parameters for Table 5.6.</td>
<td>46</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison of different $\alpha$ values.</td>
<td>46</td>
</tr>
<tr>
<td>5.8</td>
<td>Parameters for Figure 5.3.</td>
<td>46</td>
</tr>
<tr>
<td>5.9</td>
<td>Parameters for Section 5.7.</td>
<td>48</td>
</tr>
<tr>
<td>5.10</td>
<td>Parameters for Figures 5.5 and 5.6.</td>
<td>48</td>
</tr>
<tr>
<td>5.11</td>
<td>Parameters for Figure 5.7.</td>
<td>49</td>
</tr>
<tr>
<td>5.12</td>
<td>Parameters for Figure 5.8.</td>
<td>49</td>
</tr>
<tr>
<td>5.13</td>
<td>Parameters for Figure 5.10 and Figure 5.9</td>
<td>50</td>
</tr>
<tr>
<td>5.14</td>
<td>Parameters for Figure 5.10</td>
<td>50</td>
</tr>
</tbody>
</table>
The thirty spokes unite in the one nave;  
but it is on the empty space (for the axle), that the use of the wheel depends.  
Clay is fashioned into vessels;  
but it is on their empty hollowness, that their use depends.  
The door and windows are cut out (from the walls) to form an apartment;  
but it is on the empty space (within), that its use depends.  
Therefore, what has a (positive) existence serves for profitable adaptation,  
and what has not that for (actual) usefulness.

Daodejing, Verse 11 [TL91]
Introduction

In the following sections, we present the key concepts discussed in this thesis and state the goal we aim to achieve.

1.1. Problem Description

The human experience of reality comprises many layers of sensory perception, analysis, and interpretation of such, memories and imagination. Seeing an image is never pure perception, but analysis and associations naturally go along with it and allow us to deduce the context and the function of the image content. Providing additional layers to this experience allows augmenting the analysis and associations and eventually the understanding of context and function. If virtual visual layers are overlaid over the perception of reality, for example by adding sensor measurements to a live video stream, this is called augmented reality. Augmented reality (AR) applications alter the perception and thus analysis of reality by adding context dependent visual information, while preserving most of the real experience. This is in contrast to virtual reality, which focuses mostly on the artificial virtual layers of perception.

In the advent of technology enabling augmented reality, the inpainting problem has gained significance in the field of real-time applications. When aimed to enhance a users experience of reality with a virtual layer, it is often not sufficient to merely add content. The additional content can be in conflict with reality. This is the case when a real object gets replaced by a virtual counterpart, or the real object disturbs the overall scenery. Also from an artist’s perspective, removing the familiar or expected, shifts the viewers focus to the designated center of attention. Thus the ability to remove content is crucial for any elaborate augmented reality application. Altering the experience of reality by exclusively removing content is also referred to as Diminished Reality.

Image inpainting deals with the problem of completing images that have been corrupted or
where parts have been removed on purpose. The missing pixels have to be estimated such that
the image looks complete and the reconstructed areas blend with the image nicely.

A particular case of inpainting in the context of diminished reality is Object Removal. Its aim
is the reconstruction of the background occluded by the object that is to be removed. This is in
contrast to Restoration, where the aim is to fix areas missing due to noise, corruption or overlaid
text. In the case of restoration, the object itself and not the background is to be reconstructed.

The project motivating this thesis originates from the context of a museum exhibition. The
visitor aims the camera of a tablet computer at a sculpture and sees an animated equivalent of the
sculpture on the screen. Preserving the background with all its static and dynamic components
is crucial for the intended illusion. At the same time, the real physical object should not be
visible at any moment. Since the virtual equivalent of the sculpture is moving, its visual hull
is not guaranteed to overlap the real objects hull. As a consequence, the real object has to be
removed from the video stream before rendering the virtual counterpart.

1.2. Goal and Final Product

The primary objective of this thesis is to propose an algorithm solving the inpainting problem
in the context of real-time augmented reality applications. To study and verify the algorithm’s
performance and correctness, a diminished reality application is developed, which handles the
challenges imposed by the scenario described earlier. More explicitly, the desired properties of
said application are:

- The ability to run on mobile devices, supporting both Android and iOS.
- The video stream should be processed and presented in real-time.
- The resulting inpainting should be visually pleasing and consistent with the rest of the
  video.
- The application should be general purpose, no prior knowledge or restrictions on the
  scene and environment is assumed. Furthermore, the area to be inpainted should not be
  constrained in shape or size.

Our goal is to present an algorithm for the video inpainting problem that conforms to the state
of the art. Further, we wish to proof that such an algorithm can process a video in real-time on
a mobile device, provided that the computational power of the GPU is used.

1.3. Constraints

All the properties mentioned above impose a number of tight constraints, listed next:

Performance One major concern when targeting real-time processing on a mobile device
with limited computational power is the performance. Processing a single frame must take at
most a fraction of a second and includes a number of preparing steps before the actual inpainting. Since most modern mobile devices are equipped with a decently powerful GPU, leveraging that in order to achieve the required performance is an obvious choice. This allows for a high computational throughput but implies a number of limitations on the structure of the algorithm.

**Different Devices**  We aim at supporting a largest possible set of devices with differing processing power. This requires an approach that can be adapted to produce an acceptable frame-rate.

**Scene Complexity**  Not limiting the algorithm to certain classes of scenes or images prohibits heuristic simplifications of the inpainting problem or using any prior knowledge specific to the scene.

### 1.4. Our Approach

In the following chapters, we first discuss the relevant prior work in the field of image and video inpainting, especially the work of Herling and Broll and their PixMix algorithm [HB14]. Based on PixMix, we propose a new algorithm that can easily be parallelized and implemented on the GPU in Chapter 3. We formulate the inpainting problem as a minimization of a cost function, which is evaluated per pixel for all pixels that lay inside the hole. We minimize this cost function for each pixel independently using a gradient descent optimization.

In Chapter 4 we present two implementations of our algorithm. To experiment and study the algorithm, we first made a prototype implementation using Python. Once we were satisfied with the results, we implemented the entire pipeline, including the tracking and removal of a physical object from a video stream that makes use of the GPU. The results we achieved are very good in most cases. We present them in Chapter 5.
Related Work

To identify the most promising approach to realize real-time object removal, we start with an overview of the past research in the field. First, we discuss various solutions to the image inpainting and extensions to video inpainting. We also briefly discuss the problem of quality assessment in the context of image inpainting. In the second part, we evaluate the presented solutions and identify the one most suitable for our purpose. Finally, we discuss the PixMix algorithm introduced by Herling and Broll [HB14].

2.1. Image Inpainting

The inpainting problem deals with finding the colors of missing pixels in an image. An image $I$ is decomposed into two disjoint areas, one with known pixel values called the source area $S$ and one for which the pixel values are missing named target area $T$ or simply hole. Parts of the image can be missing because of the image has been corrupted, or they are intentionally removed to omit a certain fragment from the image.

A solution to this problem can only be qualified by a human observer and thus the notion of a correct solution is not strictly applicable. Moreover, there are typically several ways to solve the problem that lead to a comparable quality. Therefore, the inpainting problem is ill-posed and we need to make additional presumptions in order to have a more concise guidance when searching for the best solution.

The inpainting problem has its roots in texture synthesis, which deals with generating new texture patches that resemble some template image. A large number of algorithms to solve the inpainting problem have been proposed in recent years. These approaches can roughly be classified into two groups: The first group uses partial differential equations (PDE) to solve a diffusion equation, which describes the transport of the known pixel colors into the hole. The
2. Related Work

second identifies image patches from the source area and copies them into the target area to cover the hole.

2.1.1. Diffusion-based Algorithms

When defining a good inpainting, a possible definition is to have a smooth color transition from the source area into the hole. This can be achieved using PDEs, an approach first used in image filtering [Wei96, PM90] with the goal of removing noise or upscaling images without losing high frequency details or introducing artifacts. A second desirable property is that structures, like lines, should be extended from the border of the hole into the hole. Structures can be described as areas of constant color intensity and the direction of the smallest intensity change is called the isophote. Anisotropic diffusion has been used by Bertalmio et al. [BSCB00] to continue lines into the hole while preserving their orientation. Other approaches were inspired by PDEs used in physics to describe heat diffusion or from fluid-dynamics, for example the work of Bertalmio et al. [BBS01], which uses the Navier-Stokes equation.

While these approaches perform well on images with mostly uniform areas and structures, they cannot reconstruct texture. The diffusion suppresses high frequencies and leads to blurring.

2.1.2. Patch-based Algorithms

The following section is mostly based on A survey of the state-of-the-art in patch-based synthesis by Barnes and Zhangrnes [BZ17] which provides an excellent overview on the topic.

Another approach to the inpainting problem uses the notion of pixel patches. A patch is a small texture sample, typically only a few pixels in width and height. This approach requires a solution to the inpainting problem to only use texture material present in the image and therefore only pixel patches from the source area to fill the hole. The aim is to capture the image’s characteristic appearance and synthesize new texture that embodies these characteristics. This is achieved by copying patches from the source to the target area.

Patch-based algorithms typically incorporate two steps: Patch matching and pixel synthesis. Patch matching is the process of finding good candidate patches to fill the hole with. This is typically done by looking for patches that minimize a suitable cost function. Greedy minimization strategies start copying patches into the hole starting at the the border of the hole and move inwards shell by shell. Once all target pixels are covered by a patch, the algorithm terminates. Iterative strategies start with an initial matching and then repeatedly keep looking for better patches. No matter what strategy is used, matching patches always includes searching for a good candidate. The naive approach is to perform an exhaustive search on all candidate pixels and pick the best. However, this is very slow and thus only feasible for small images. Other approaches like PatchMatch [BSGF10] by Barnes et al. randomly try patches and once a suitable candidate is found, this information is propagated to neighboring patches. Locality sensitive hashing applies binning on the patches using a hash function, which captures similarity of patches. Korman and Avidan [KA11] use this strategy to speed up PatchMatch. Another technique uses tree data structures. A dimensionality reduction is applied to the patches and
2.2. Video Inpainting

these are then inserted into a space-partitioning data structure, for example a *kd-tree*, which can be to perform a nearest neighbor lookup in the patch appearance space.

The source of patches can be extended to more than one image. Using entire image collections can yield better patches than a single image. Hayes and Efros [HE07] used a collection of millions of photographs. These images were first grouped according to their scene semantic using the context-based place and object recognition system [TMFR03] proposed by Torralba et al. Finding a patch is composed of first finding the scene group, the input image belongs, and then looking for good patches within that group.

Patch-based approaches are capable of reproducing texture accurately. The problem formulation using a cost function allows crafting a large variety of desired properties of the resulting inpainting. Patch-based approaches typically outperform diffusion-based techniques in most scenarios, especially when the hole size is more than a few pixels wide and texture becomes crucial. However, if a structure needs to be reproduced, which is not available as a patch in the patch search space, diffusion techniques can outperform them [GM14].

2.1.3. Hybrid Methods

Bertalmio et al. [BVSO03] proposed a hybrid technique that combines the strength of reconstruction texture of the patch-based approaches with the ability to continue structures into the hole of diffusion techniques. The image is split into a texture and a structure layer, which when overlaid result in the original image. A patch-based approach used for the texture layer and a diffuse for the structure layer. Both results are then recombined.

2.2. Video Inpainting

Video inpainting is the extension of image inpainting onto a collection of subsequent video frames. In comparison to image inpainting, the temporal order of the frames can be used to guide the process. If this is done off-line, the entire collection of frames is available when searching for good patches.

The work done by Granados et al. [GKT+12] combines frames, which show a part of the background behind the object they want to remove. These frames are combined and the true background is restored. In a first step, objects with a similar scene depth are tracked across the frames by estimating the homographies describing the transformation from one frame to the next. They assume these objects to be mostly planar, which allows to get depth information without having to reconstruct the entire 3D scene. For each target pixel, they identify a frame that shows the background for this pixel, as it would be seen from the perspective of the image to be inpainted.

Another approach to video inpainting proposed by Hocking et al. is *Guidefill* [HMS16]. They address the problem of 2D to 3D conversion of movies. To produce a stereoscopic experience, the scenes must be rendered from a second perspective. Since this is an ill-posed problem, areas without information from the original frames must be reconstructed. These are generally small
2. Related Work

gaps and using a diffusion-based technique that transports colors along guiding splines, similar to the isophotes, produces satisfying results.

PixMix [HB14] by Herling and Broll is another approach, which introduces tracking of object contours for object removal in a video. Similar to [GKT+12], they compute the homographies describing the transformation of the object contours from one frame to the next. They operate on the mapping between source and target patches and forward a good mapping of patches to the next frames using the information from the object tracking. PixMix also introduces a novel image inpainting approach based on patch matching. This is discussed in depth in Section 2.6.

2.3. Inpainting Quality Assessment

It is desirable to have a measure to assess the quality of a result produced by any inpainting pipeline. Not only would this make the comparison of results easier and less biased, but it could also be used as cost a function for an optimization approach to the inpainting problem.

We could compare the error of the pixel colors of the reconstructed background using the original image as ground truth. However, when want to remove an object, we do not expect the reconstructed background to be similar to the ground truth. The ground truth contains the object we want to remove in the reconstructed area. We want this area to be similar to the background surrounding the object.

Guillemot et al. describe the problem of quality assessment accurately:

"The quality assessment of inpainted images is another open and difficult issue, as no quantitative metrics exists. Fidelity metrics cannot be used given that, for most inpainting applications, the ground truth is in general unknown. One has to rely on a subjective assessment to evaluate whether the inpainted images are visually pleasing and physically plausible." [GLM14, p. 143]

Therefore, mostly the human perception is decisive for quality assessment.

2.4. Approach and Rationale

There is a wide range of possible approaches to the inpainting problem. The following section provides a comparison of different classes of approaches, aiming to identify the one best fit to cope with the constraints discussed in Section 1.3.

If not explicitly mentioned, the approaches introduced have (to the best of our knowledge) never been implemented and are only discussed here as comparison to other proposed methods.

For this discussion, we assume that the position of the camera with respect to the real physical object to be removed is given, as well as the area to be inpainted. For more on how this is done, see Section 4.1.2.
2.4. Approach and Rationale

2.4.1. Modeled, Learned or Synthesized Background

An intuitive solution to the object removal problem is using knowledge of the true background. This can be acquired in various ways, as described next.

**Full prior knowledge** In case of the example of the museum exhibition mentioned in the introduction, a 3D scan of the exhibit hall can be made. The problem is then reduced to tracking the camera’s position in the hall and rendering the missing areas using the scan. ¹

**Pros:** Given the 3D scan and the spatial tracking, the inpainting itself is trivial.

**Cons:** It can only deal with static content. Other visitors, a change of exhibits in the background and the light direction cannot be reproduced without changing the reconstructed scene. Furthermore creating an accurate scan can be very time-consuming.

This violates the property of being general-purpose and is thus not an option for this project.

**Partial Prior Knowledge** Some static properties of the scene can be used to guide the inpainting process.

One suggestion is to model the scene using approximating planar surfaces only. This and the knowledge of the relative position of the camera to the physical exhibit provides an estimate of the distance between the camera and the true background. Like a simplified SLAM² procedure, the true color of the background can be deduced from previous frames. Using the depth information of the scene, the background pixels to be reconstructed can be identified in previous frames where the exact background was not occluded. Given enough frames, the entire background can be deduced. By discarding older frames, an adaption to changes to the scene is possible, as long as the depth of the background objects does not change.

**Pros:** In the best case, the true background is reconstructed and after some time the inpainting can be robust and consistent across frames. Minor changes in the background can be handled.

**Cons:** Background with dynamic depth, such as other museum visitors, will not be reconstructed at all, or as part of a background plane, depending on their degree of movement and the number of past frames available for reconstruction.

With regards to the targeted ability to adapt to any setting, acquiring the depth information of the environment is not an option. This approach depends less on the accuracy of the scene model and its creation is far more simple than the full 3D scan mentioned in the approach discussed first. However, it still violates the general-purpose principle.

**Learned Background Without Prior Knowledge** Research in *Computer Vision* has lead to a large number of methods capable of reconstructing 3D maps based on a collection of images. In the domain of real-time processing, SLAM algorithms are used to learn a sparse 3D

---

¹A similar approach was used for a VR art exhibition [AG]. The halls were modeled and altered, allowing a fictional glance into an apocalyptic future of the exhibit hall.

²Simultaneous localization and mapping
2. Related Work

representation of the environment [NLD11]. Once such a 3D representation is present, full knowledge of the backgrounds depth is available and the inpainting problem can be solved in the same manner as with full or partial prior knowledge. The quality of the SLAM algorithm, however, is highly dependent on a sufficiently large distribution of the camera position and orientation when acquiring the images. If this is not given, the reconstruction is distorted, incomplete or not successful at all.

The method by Granados et al. [GKT+12] discussed in Section 2.2 makes use of the fact that the true background can be learned without having to estimate a dense depth representation. However the processing time is in the dimension of hours for a small number of frames and the full video, including future frames, is used for the inpainting of any frame in the video.

**Pros:** The true background can be reconstructed without any prior knowledge. The result is consistent across frames.

**Cons:** The background occluded by the object to be removed must be static for as long as the learned depth estimate or homographies are used. Also the background has to be fully visible at least in one frame.

These limitations are in conflict with the desired property of being able to deal with any kind of scenario, including fast moving objects.

In a museum exhibition, the visitor using a tablet computer to examine an augmented sculpture is not likely to move much sideways, but rather to rotate the tablet and move it back and forth to center the sculpture on the screen. These are conditions under which SLAM is likely to fail due to the narrow baseline distance, and unlikely to produce a frame showing the full background.

Despite these limitations, the first approach, inspired by the SLAM problem, could significantly benefit from this scenario. This is because the sculpture is static and its relative position to the camera is known.

**Synthesized Background**  The naive solution to video inpainting is to treat each frame as an independent image inpainting problem. No assumptions on the setting or the background must be made. Moreover, most image inpainting algorithms are not concerned with the concept of fore- and background at all. Discarding the third dimension entirely seems like a severe loss of potentially useful information. However, since the video is an ordered collection of images, it does not truly contain any information about the third spatial dimension, but much rather about the temporal dimension. As discussed, extracting the third spatial dimension is not trivial, so focusing on the temporal dimension instead is a reasonable way to exploit the full potential of the video data.

Additionally to the video, the relative position of the camera to the sculpture is known. This provides additional spatial information, but not the depth of the background, which would simplify the problem considerably. Compared to the homographies used by Granados et al. [GKT+12], this tracking information is also of lesser quality. These homographies describe the relation of background objects across frames, where the tracking contains no information on the backgrounds geometry whatsoever. However, the tracking information is very similar to the frame to frame contour tracking used by Herling et al. in their PixMix approach [HB14]. For tracking the object to be removed, they first compute the homographies using motion detection on selected
contour points and apply some refinement to generate the inpainting mask. These homographies are additionally used on multiple occasions throughout their video inpainting pipeline, most prominently when guiding the inpainting of the next frame. This will be discussed in more detail in Section 2.6.3.

**Conclusion**  It appears useful to leverage the structure of the video as sequence of frames combined with the known spatial transformations of the object to be removed from one frame to the next. In this thesis we aim to find a representation that can use these transformations to transfer the inpainting from one frame to the next. PixMix [HB14] provides valuable insight in how to achieve this and will be discussed thoroughly in Section 2.6.3.

### 2.4.2. Diffusion or Patches

As discussed in the previous chapter, the two major classes of inpainting algorithms are the ones based on diffusion and the ones based on copying patches. We evaluate which of the two is better for our purpose in the following section.

**Qualitative Comparison**  In their informative overview on a broad variety of inpainting algorithms, Guillemot and Le Meur [GM14, p. 141] state:

"Diffusion introduces smoothing and blurring artifacts in the synthesized region. To recover the texture of the hole, examplar-based methods [...][GTME13], methods using sparse representations [...][XS10], or solutions combining structure diffusion and examplar-based texture recovery as, e.g., [...][BBCS10] are more appropriate."

They propose to use diffusion based algorithms only for small holes, whereas for medium to big sized holes exemplar-based or hybrid methods are preferred [GM14, p. 140]. Their reference images for the case of object removal is reprinted in Figure 2.1. All patch-based or hybrid approaches (Figures 2.1c to 2.1f) perform clearly better than pure diffusion (Figure 2.1b).

Similarly in their evaluation, Vreja et al. [VB14, p. 10] conclude:

"The tests have shown that inpainting algorithms involving diffusion operations perform well for structural features images but cannot successfully rebuild textures."

Hybrid techniques as a combination of diffusion and patch-based approaches utilize the best of both worlds and are thus expected to be at least equally good as pure diffusion or patch-based approaches in any given scenario. However, as a consequence, they are inherently more complex and require more processing time.

Patch matching is the major challenge with patch-based algorithms. In Section 2.1.2 a number of techniques have been discussed using random search, coherence, tree data structures or hashing or more efficiently a combination of these, such as PatchMatch [BSGF10] and CSH [KA11]. As algorithms making use of traversing trees and looping are inherently unfit to be executed on the GPU. We will discuss the technical details of the limitations of GPU computing in Section 4.1.1. However, since the overall objective is to minimize a distance term, any other class of minimization strategies can be used if the problem can be formulated appropriately.
2. Related Work

Figure 2.1: Object removal application (a) mask and inpainting results with methods from different categories, (b) anisotropic diffusion [Tsc06], (c) examplar-based with LLE [GTME13], (d) patch sparse representation [XS10], (e) hybrid with one global energy minimization [BBCS10], and (f) patch offsets [HS12]. Reprinted from [GM14] (With images provided to Guillemot and Le Meur as courtesy by the respective authors and (a) courtesy of www.magazinehive.com)

Iterative or greedy strategies have been proposed for solving the patch matching problem, as stated in Section 2.1.2. The algorithm has to be compatible with a large number of devices, which may drastically differ in their computational power. Thus, the algorithm has to be able to generate results within a computational budget that might be as low as a few hundred milliseconds on a tablet computer. Choosing an iterative approach allows us to adjust the number of iterations without leaving any pixels without inpainting, merely reducing the quality. This is preferable over a greedy approach that might not complete its computation within the computation budget.

The patch size and density is a final feature to be considered. While most strategies define a patch as a statically sized and shaped neighborhood of pixels, Herling and Broll propose a relaxation of the definition of a patch in their PixMix paper [HB14]. They match single pixels instead of entire patches, but use a cost function that encourages neighboring pixels to be mapped to patches. This relaxed definition of a patch can easily integrate rotation and scale invariance by simple modifications of the cost function.

Additionally, the mapping information computed on a downscaled version of the image, can be reused at higher resolutions and further refined. At low resolution, the neighborhood covers a large area, identifying low-frequency features, while at high-resolution finer details are considered. Furthermore, the mapping information computed for one frame can serve as the initialization for a next frame, if properly transformed.

To conclude, patch-based approaches outperform diffusion techniques in terms of quality and hybrid techniques in terms of performance. Whether the qualitative superiority of hybrid tech-
niques leads to a significant improvement of the result in the context of this projects, can not be answered in this thesis. However, initially choosing a patch-based approach to start with does not exclude a hybrid extension, in case the performance permits it.

An iterative approach helps to adapt the computational cost to the given budget. A relaxed definition of patches allows for efficient reuse across different resolutions and frames.

2.5. Inpainting problem as Optimization

Formally, inpainting aims at completing an image $I$ that is composed of a source region $S$ with known pixel values and a target region $T$ with missing pixel values. $I$ is the disjoint union of $T$ and $S$. $T$ is also referred to as hole. Formally it can be described as the optimization problem of finding a mapping function $f : T \rightarrow S$ that globally minimizes the cost function $cost_{tot}(f) = \sum_{p \in T} cost(p)$. Where $cost(p)$ analytically defines the quality of the mapping of an individual pixel $p$ with respect to a subset of the image’s pixels. In the context of image inpainting, defining quality is not straightforward, since it is hard to tell which inpainting is best. Commonly a resulting inpainting is rated by how easily the inpainted area can be noticed and whether its texture looks plausible and consistent with the known part of the image. As mentioned in Section 2.3, some effort has been made towards automatic quality assessment, but the results cannot be easily used in order to define an adequate cost function.

2.6. The PixMix Algorithm

In comparison to the previous work in the field of real-time video inpainting, the PixMix paper [HB14] by Herling and Broll excels not only with visually appealing results, but by introducing an algorithm that only requires minor changes in order to be implemented on the GPU. Since the work discussed in this thesis is strongly based on PixMix, the following section will present an introduction to the aspects of PixMix relevant to the understanding of the the discussion of our approach in Section 3.1. We will introduce the key concepts here, the entire algorithm adapted to this project’s needs is presented later in Section 3.2.

2.6.1. Cost function

Following the formal definition of the inpainting problem given in Section 2.5, Herling and Broll suggest a cost function composed of multiple terms. Relevant to this discussion are the color $cost_{color}$ and spatial $cost_{spatial}$ error terms.

$$cost(p) = \alpha cost_{color}(p) + (1 - \alpha)cost_{spatial}(p) \quad (2.1)$$

$\alpha \in [0, 1]$ is used to weight the two error terms. This weight factor allows to balance the impact of the two terms, to make sure that a single term cannot dominate the entire optimization.
2. Related Work

**Spatial cost**  While the mapping is defined by individual pixels and not using patches, the aim is still to reproduce texture by copying entire patches from the known part of the image. The spatial cost should enforce that all neighbors of $p$ are mapped to neighbors of $f(p)$. This encourages extensible patch-like structures without enforcing a static patch. If the mapping has a piecewise patch-like structure, texture from the source region $S$ can get reconstructed inside the hole. This is formally captured using the following definition of the spatial cost term:

$$cost_{\text{spatial}}(p) = \sum_{\vec{v} \in N_s} d_s[f(p) + \vec{v}, f(p + \vec{v})] \omega_s(\vec{v}),$$  \hspace{1cm} (2.2)

where $N_s$ is a neighborhood and its elements $\vec{v} \in N_s$ are the offsets from the neighborhood’s center. The shape of the neighborhood is discussed in Section 4.1.6. $d_s$ is a robust distance function. $\omega_s \in \mathbb{Q}$ is a weight parameter with $\sum_{\vec{v} \in N_s} \omega(\vec{v}) = 1$. The robust distance function and the rational for using it is discussed in Section 3.1.2.

**Color cost**  Regarding the color, two properties must hold in order to get credible inpainting: First, the seams of neighboring patches, represented as a neighborhood $N_c$ around the pixel, must not be visible. Second, structures must be continued through such a patch. This can be achieved by making sure that the neighborhood around $p$ looks similar to the neighborhood around $f(p)$. If structures are present in the neighborhood around $p$, a good match would be a part of the image that contains a similar structure. Since neighborhoods of neighboring pixels have a large overlap, they share most neighbors and have thus similar constraints imposed by their neighbors’ colors. This is formalized by the following equation:

$$cost_{\text{color}}(p) = \sum_{\vec{v} \in N_c} d_c[i(p + \vec{v}), i(f(p) + \vec{v})] \omega_c(p + \vec{v}),$$  \hspace{1cm} (2.3)

where $N_c$ is a neighborhood, $d_c$ a pixel intensity distance measure, $i(p)$ the color value of a pixel $p$ and $\omega_c$ equivalent to $\omega_s$.

2.6.2. Image inpainting

Herling and Broll introduce a solution to the image inpainting problem. Its relevant aspects are highlighted in the following section.

Like most patch-based image inpainting algorithms, PixMix has two major stages:

1. Finding a good mapping
2. Synthesizing the pixel colors for all pixels inside the hole

While the second step only requires looking up the pixel color using the given mapping, the first step is more involved. It requires finding the mapping that synthesizes the lowest cost. The authors suggest an approach similar to the PatchMatch\[BSGF10] algorithm. Once the mapping function is initialized, a source pixel is mapped to a random target pixel. If this new mapping results in a lower cost then before, it is kept, otherwise the old mapping is preserved. If a better mapping is found, it gets propagated to the neighboring pixels. This is repeated until a sufficiently good mapping function $f$ is found.
Image pyramid  In computer graphics, an image pyramid is a data structure that contains an image and scaled-down versions of this image with different resolutions. They are also called mipmaps and are useful if multiple resolutions of an image are needed.

The optimization is applied on such an image pyramid \( L_0, \ldots, L_m \), where the highest resolution level \( L_0 \) is the original image and the lower level scaled-down versions of the image. Starting at the lowest level \( L_0 \) the next higher level is created by downscaling by a factor of two, creating an image half the width and height of the previous level. The optimization of the mapping function \( f \) starts at low resolution and once a sufficiently good candidate for \( f \) is found, this is then forwarded to the next higher resolution level until the final level \( L_0 \) is reached. The benefit of using image pyramids is threefold: First, the optimization is less likely to get stuck in a bad local minimum, because at low resolution the cost function with respect to \( f \) has a smaller number of local minima. As a consequence, the total cost synthesized by the final mapping is lower. Second, structures of various frequencies are captured at the respective resolutions. The neighborhoods \( N_c \) and \( N_s \) cover a larger image area at a lower resolution and are thus sensitive to lower frequency structures. As the mapping gets passed to the next pyramid level, the structures implicitly remain present in the mapping function and are later reproduced during the image synthesis. Third, the convergence is sped up. A single iteration on the full resolution image is much more computationally expensive than on lower resolutions. Presenting a good initialization to the high-resolution levels can thus significantly speed up the convergence.

2.6.3. Video inpainting

Additional to static image inpainting, this section discusses the contributions of PixMix to video inpainting, as far as they are relevant to this thesis.

The authors introduce a guided object contour selection and tracking across the video frames based on the video frames exclusively. They explicitly compute the homographies mapping the object contour from one frame to the next, whereas for this project the transformations are provided by image tracking.

The video inpainting pipeline is presented as follows: The first frame is inpainted according to the method described above. The resulting mapping \( f_n \) function is transformed to the next frame using the corresponding homography \( H \) and then used as initialization for the optimization of the next frames mapping \( f_{n+1}^{init} \). This transformation of the old mapping to the new frame is given by \( f_{n+1}^{init} = H^{-1}f_nH \). The same optimization as before is then performed using this initialization.

In addition, the authors suggest using a keyframe that acts as reference model. This reference model is used such that the inpainting of a sequence of frames remains coherent to the key frame. This is achieved adding another term to the cost function.

\[
\text{cost}'_{\text{color}}(p) = \sum_{\overrightarrow{v} \in N_c} d'_c[i(p + \overrightarrow{v}), r(p + \overrightarrow{v})] \omega'_c(p + \overrightarrow{v}),
\]

where \( r(p) \) is the pixel value of \( p \) in the reference model, \( d'_c \) and \( \omega'_c \) distance and weight function as described for the color cost. This additional cost term forces a pixel to be similar to its corresponding pixel in the reference model, preventing drastically differing results for similar
2. Related Work

frames. \( r(p) \) is determined by applying the backward transformation from the current frame to the key frame using the computed homographies.
Method

Based on the algorithm introduced by Herling and Broll in their PixMix paper [HB14], we propose an algorithm suitable for a GPU implementation. We also present an overview of the entire algorithm in the second part of this chapter.

3.1. Our Approach

Though strongly leaning on the PixMix algorithm for guidance, we introduced some changes before we implement the algorithm on the GPU.

Most important is the introduction of a different optimization method and, as prerequisite, converting the discrete nature of the image inpainting problem into a continuous one. Further, we propose a normalization for the cost function, discuss the use of a robust distance function and show how the gradient of the cost function can be computed. We also discuss how these modifications can lead an invalid mapping and how to deal with that.

3.1.1. Optimization

Our initial observation is that the cost function is differentiable. We augment the discrete image pixel domain by applying bilinear interpolation to the image, and the discrete color sampling function $i$, as well as the mapping function $f$, to continuous functions.

Given this continuous nature, a large set of optimization methods is available. We use gradient descent with a carefully considered step size.

The gradient of the cost function can be computed on each pixel in parallel and the mapping if updated locally using this gradient. This makes our approach a local optimization with every
3. Method

local update, affecting all neighboring pixels during the next iteration. Because the pixels are highly connected through these neighborhoods, no local optimization step is independent of the others.

Spatial Cost Term

The spatial error term has been altered to make it more suitable for the gradient descent optimization and to further relax the patch structure by dropping the implicit angular constraint present in the original formulation. The definition from PixMix requires the top right neighboring pixel of \( p \) to be mapped to the top right neighbor of the \( f(p) \). This explicit spatial structure is relaxed by only requiring the distance to be the same. Note that this only leads to rotation invariance, if we change the color cost term accordingly, which we did not implemented for this thesis.

We observe a drift of the mapping towards the edge of the hole caused by the spatial cost term. This is because the spatial cost term is trying to preserve neighborhoods to create patch-like structures. As a consequence, the mapping of a pixel with a source pixel in its neighborhood converges to the neighborhood of that source pixel. Even though only target pixels close to the edge of the hole have source pixels in their neighborhood, their mapping will pull all surrounding pixels’ mapping towards the edge as well, as the optimization proceeds. Therefore, we limit the neighborhood \( N_s(p) \) around \( p \) to the target pixels.

Moreover, we set the weight parameter \( \omega_s = \frac{1}{||N_s||} \).

This leads to the following adapted definition of the spatial cost term:

\[
\text{cost}'_{\text{spatial}}(p) = \frac{1}{||N_s||} \sum_{\overrightarrow{v} \in N'_s} c[d_s[f(p), f(p + \overrightarrow{v})], ||\overrightarrow{v}||],
\]

(3.1)

where \( N'_s = N_s \cap T \) is the neighborhood reduced to the target pixels and

\[
c[a, b] = \begin{cases} |b - a|, & \text{if } |b - a| \leq \tau \\ \tau, & \text{otherwise} \end{cases}
\]

(3.2)

is the absolute norm clamped by \( \tau \). This \( \tau \) is the same as discussed in Section 3.1.2. As consequence, the neighboring pixel can be at any position on the sphere of radius \( ||\overrightarrow{v}'|| \), but since this must hold for any potential neighborhood, the lowest error is still the same as for the definition by Herling and Broll. The above definition of the cost function will be referred to as \( \text{cost}_{\text{spatial}} \) and the above \( N'_s \) as \( N_s \) from here on.

Gradient

In this section, we discuss how the gradient of the cost function is built, as it constitutes the core part of the algorithm. Since the color cost function is a linear combination of the color and the spatial cost terms, its gradient is a linear combination of the gradient of these two terms.

\[
\nabla \text{cost}(f) = \sum_{\forall p \in T} \nabla \text{cost}(p) = \sum_{\forall p \in T} \alpha \nabla \text{cost}_{\text{color}}(p) + (1 - \alpha) \nabla \text{cost}_{\text{spatial}}(p).
\]

(3.3)
3.1. Our Approach

**Spatial Cost Gradient**  The spatial cost term aims at keeping a pixels mapping \( f(p) \) at the proper distance of its neighbor’s \( p_n = p + \vec{v} \) mapping. That is

\[
d_s[f(p), f(p + \vec{v})] = ||\vec{v}|| \quad \forall \vec{v} \in N_s.
\]

(3.4)

The gradient is the vector pointing towards the point that satisfied the above equation best.

\[ \Delta s = f(p + \vec{v}) - p' \]

\[
\nabla cost_{\text{spatial}}(p, p') = \sum_{\vec{v} \in N_s} \Delta s \frac{d_s[\Delta s] - ||\vec{v}||}{||\Delta s||}
\]

(3.5)

\[
= \sum_{\vec{v} \in N_s} [(f(p + \vec{v}) - p') \frac{d_s[f(p + \vec{v}) - p'] - ||\vec{v}||}{||f(p + \vec{v}) - p'||},
\]

(3.6)

where \( p' \) is the candidate source pixel or the previous iterations mapping \( f_{n-1}(p) \)

**Color Cost Gradient**  The color cost uses the pixel color values as input that is not analytically differentiable. Instead, the finite difference approximation is used to compute the gradient of the color cost \( \nabla cost_{\text{color}}(p) \) for a pixel \( p \). The gradient is then

\[
\nabla cost_{\text{color}}(p) = (\nabla_x cost_{\text{color}}(p), \nabla_y cost_{\text{color}}(p),
\]

(3.7)

where

\[
\nabla_x cost_{\text{color}}(p) = \frac{cost_{\text{color}}(p + [\epsilon, 0]) - cost_{\text{color}}(p - [\epsilon, 0])}{2\epsilon}
\]

(3.8)

\[
\nabla_y cost_{\text{color}}(p) = \frac{cost_{\text{color}}(p + [0, \epsilon]) - cost_{\text{color}}(p - [0, \epsilon])}{2\epsilon}
\]

(3.9)

where \( \epsilon > 0 \) is set to 0.5 throughout the rest of this discussion. This guarantees that we do not over- or undersample the image by following the Nyquist-Shannon sampling theorem.

**Step Size**

Gradient descent algorithms are inherently sensitive to overshooting if the magnitude of the gradient is too large. To avoid overshooting and to speed up the convergence, the gradient typically gets scaled or clamped.

The color error, by definition, is bound by \( d_c[\text{white}, \text{black}] \) where white and black are pixels with the corresponding colors. So it can be easily normalized to \([0, 1]\), as shown in Section 3.1.2. Since computing the color cost gradient uses finite differences with a total distance \( 2\epsilon = 1 \) pixel in between the sampling, the gradient can never exceed \([\text{max}_c, \text{max}_c] \) where \( \text{max}_c \) is the maximum possible color cost. After the normalization to \([0, 1]\) introduced in Section 3.1.2, this yields that \( \nabla cost_{\text{color}} \in [0, \sqrt{2}] \).

As we will discuss in Section 3.1.2, the spatial cost term is normalized to be at most 1 and correspondingly the gradient of the spatial cost term is normalized to be at most of length 1.
3. Method

(a) The spatial cost term plotted against the color cost term. Each dot represents a pixel with the y axis being the value of the spatial cost term and x the color cost term. The data is taken from a single optimization step.

(b) Average of the total cost term for 22 iterations of optimization.

Figure 3.1.: Spatial cost term, color cost term and total cost.

The total cost is a weighted sum of the spatial and color cost using the blending parameter $\alpha$. Considering the definition in Equation (2.1), the total cost has its maximum value of $\sqrt{2}$ with $\alpha = 1$. When the additional correction of the color cost term discussed in Section 3.1.2 is applied, this maximum can even be higher.

Again, the Nyquist-Shannon sampling theorem tells us that a step size of at most 0.5 pixels is ideal to avoid undersampling of the image. We, however, want to allow for slightly higher step sizes for fast conversion, if a mapping is very far from a good state.

Experiments with the Python prototype have shown that the applied normalizations and clamping of the gradients lead to a step size of around 0.5 pixels with a few higher outliers. After a few iterations the step size typically drops drastically and only a few pixels. Figure 3.1b shows the total cost for 22 iterations on a high-resolution level. The cost never exceeds 0.23 and drops drastically after 10 iterations. In Figure 3.1a every dot is a single target pixel, the y axis is the value of the spatial cost term and x the color cost term of the last step of this levels optimization. We used these distributions to verify that the step size was mostly below 0.5 pixels and not drastically below and also to study the impact of the $\alpha$-weight.

3.1.2. Other Modifications

Experiments with the Python prototype (see Section 4.2) have lead us to introduce normalization of the cost and spatial cost term. We discuss these normalizations and the robust distance function used in the spatial cost term in the following section.
3.1. Our Approach

Robust Distance Function

The definition of the spatial cost term in Equation (2.2) introduces a robust distance function \( d_s \). The capped L2 norm is used for \( d_s \).

\[
d_s(p, p') = \begin{cases} ||p' - p||, & \text{if } ||p' - p|| \leq \tau \\ \tau, & \text{otherwise} \end{cases},
\]

(3.10)

where \( \tau > 1 \) is a clamping threshold in pixel unit and \( || \cdot || \) the L2 norm.

In the following discussion we assume \( d_s(p, p') = ||p' - p|| \) for small distances. The above definition is minimal if \( ||f(p + \overrightarrow{v}) - p'|| = ||\overrightarrow{v}|| \forall \overrightarrow{v} \in N_s \). If \( ||f(p + \overrightarrow{v}) - p'|| > ||\overrightarrow{v}|| \) the mapping \( f(p) \) is too far away from \( p \) and must be pulled closer. In case of \( ||f(p + \overrightarrow{v}) - p'|| < ||\overrightarrow{v}|| \) \( f(p) \) is too close to \( p' \) and must get pushed away. When \( f(p + \overrightarrow{v}) = p' \), the above equation is undefined. If this happens, the contribution of this neighbor to the spatial cost can be ignored. This strategy assumes that this case is rare and in the course of the optimization this case will be resolved quickly.

Spatial Cost Normalization

The robust distance function discussed in the previous paragraph has a significant impact on the maximum possible value of the spatial cost term. This maximum is proportional to \( \tau \), and so is the contribution of the spatial cost term to the total cost term. We divide the spatial cost term with \( \tau \) to normalize it to \([0, 1]\). Due to this modification the range of the spatial cost term is independent of \( \tau \).

Color Cost Normalization

We normalize the color cost term to \([0, 1]\) to make it independent from the color format used. In addition to that, the overall contrast of the image has a significant impact on the color cost’s magnitude. Images with little contrast result in a color cost that is persistently smaller than for the same image with higher contrast. To counteract that, the standard deviation of the image’s colors is computed for every frame first and the color cost then divided by its norm. This results in the following definition:

\[
cost'_{\text{color}}(p) = cost_{\text{color}} \cdot \frac{1}{||c_{\text{max}}|| ||c_{\text{std}}||} \sum_{\overrightarrow{v} \in N_c} ||i[p + \overrightarrow{v}] - i[f(p) + \overrightarrow{v}]|| \omega_c(p + \overrightarrow{v}) \cdot \frac{1}{||c_{\text{max}}|| ||c_{\text{std}}||},
\]

(3.12)

with \( c_{\text{max}} \) indicating the pixel value for the color white and \( c_{\text{std}} \in \mathbb{R}^3 \) the standard deviations of the RGB color channels. Note that the alpha channel is ignored and the L2 norm of the color differences computed per color channel is used as pixel intensity distance measure \( d_c \). The above definition of \( cost'_{\text{color}} \) will be referred to as \( cost_{\text{color}} \) in the following.
3. Method

3.2. Algorithm

This section describes the complete algorithm in detail. It consists of an initialization of the mapping that performs an exhaustive search, then an iterative optimization of the color and spatial cost simultaneously. The initialization is performed on a low-resolution mipmap level and the optimization iterates on each level until convergence and then moves on to the next higher resolution. Since the cost is optimized for each pixel independently using gradient descent, no guarantee on the convergence can be given. It is a local optimization, hoping to find at least a good local minimum and reducing the risk of getting stuck in a bad one by carefully guiding the process from low-resolution to high-resolution.

3.2.1. Initialization

The initialization is performed on the highest mipmap level at which enough structure is already present. If the resolution is too low, the color cost cannot capture the structural characteristics of the image. If it is too high, the initialization will be very slow. Starting at a resolution with a width of 16 pixels has proven to work best.

For all known pixels the mapping is set to point to itself, \( f(p') = p' \forall p' \in S \). The unknown pixels \( p \) are initialized by exhaustively searching all potential target pixels \( p' \) for the one generating the smallest color cost \( f(p, p') \). That is \( f(p) = \arg\min_{p' \in S} \text{cost}_{\text{color}}(p, p') \forall p' \in S, p \in T \).

A source pixel \( p \) can be initialized only if there are enough pixel in its neighborhood to robustly compute the color cost. For this reason, the initialization consists of multiple iterations. Initially, the pixels at the edge of \( S \) with enough valid neighbors are mapped. The initialization then proceeds inwards, shell by shell, until all pixels are mapped. The minimum number of mapped neighbors depends on the neighborhood size and is discussed in Section 4.1.6.

Note that we only use the color error during the initialization. Firstly, because the spatial error is not defined unless there are mapped target pixels in the neighborhood. Secondly, it is desirable for the initial mapping to be widely spread. The spatial cost will pull together pixels to patches during the optimization, so having a larger variety of good candidates reduces the risk of getting stuck in a local minimum.

However, according to the Nyquist-Shannon sampling theorem, this is under-sampling the image pixel space. As a result, we do not capture the entire frequency spectrum as desired. More on this in Figure 5.8. Instead, a grid of density 0.5 pixels, that samples interpolated pixel color values, should be used.

3.2.2. Optimization

The optimization performed on a given mipmap level consists of iteratively computing the gradient of the cost function and updating the mapping in its negative direction. The following describes a single iteration of the optimization. We assume the mipmap level and the mapping function \( f_{n-1} \) from the last iteration or initialization given.
3.2. Algorithm

The gradient of the spatial cost term can easily be computed using the derivation from Equation (3.5), the robust distance function defined in Equation (3.10) and the normalization by dividing the spatial cost by \( \tau \), as described in Section 3.1.2. Computing the color cost gradient is done using a finite difference approach, as described in Equation (3.8) including the normalization introduced in Equation (3.11). Equation (3.11) deals with the case of having a pixel with an invalid mapping in either the source or target neighborhood. The final gradient is then built as

\[
\nabla \text{cost}(p) = \alpha \nabla \text{cost}_{\text{color}}(p) + (1 - \alpha)\nabla \text{cost}_{\text{spatial}}(p)
\]

and finally the mapping is updated by adding the negative gradient to the last estimate of \( f \).

\[
f_n(p) = f_{n-1}(p) - \nabla \text{cost}(p).
\]

If the new mapping \( f_n(p) \) is not within the image borders, it gets clamped to the nearest edge pixel. The mapping is then also checked to not point into the source area, a case discussed in the following section, and then eventually stored as result of this iteration. Note that the new value is first used in the following iteration, even if neighboring pixels have not been processed yet. During a single iteration, all pixels share the same mapping function.

Mapping Back Into the Hole

Since the proposed gradient descent approach does not test and assign potential better mappings, but gradually moves them to a place of low cost, a new mapping might end up pointing into the hole, leading to an invalid mapping.

We can handle this case in various ways and will discuss three of them. An evaluation of how they affect the resulting inpainting is given in Section 5.8.

**Force** Whenever a pixel \( p \) is mapped into the hole, it gets pushed outwards by a force in direction \( v_{\text{out}} = f(p) - c \), where \( c \) in the center of the hole. We compute the center of the hole as the projection of the center of mass of the 3D model or manually select it for static test images. The resulting mapping is then \( f_{\text{corr}}(p) = f(p) + v_{\text{out}} \), which, however, might still be inside the target area. To escape the hole more quickly \( v_{\text{out}} \) is scaled to be larger, if closer to the center, and smaller, if closer to the edge of \( T \) as follows \( v'_{\text{out}} = v_{\text{out}}/||v_{\text{out}}||^2 \).

This is a simple approximation of the projection of the original mapping onto the border of \( T \), with the advantage of staying as close to the original mapping as possible. However, additional to not having any guarantee of escaping the hole within one iteration, the chance of getting mapped back into the hole in a subsequent iteration is rather high.

**Neighbor with Smallest Error** Instead of trying to stay close to the original mapping, the pixel gets remapped to a potentially good pixel anywhere in the image. Potentially good with respect to the spatial cost function means close to the neighbors mapping. Therefore, we use the neighbors’ mappings as a starting point, from which we add a random offset vector \( \vec{v} \) of length, creating eight candidate pixels when using the eight-neighborhood. For these candidates, we compute the cost and choose the one that synthesizes the lowest cost. More formally this is

\[
f_{\text{corr}}(p) = \arg\min_{p' + \vec{v}} \text{cost}(p, p' + \vec{v}) \with N \]

with \( N \) as the eight-neighborhood.
3. Method

Starting with a set of candidates generating a low spatial cost and picking the best establish a fair chance of finding a good match and being further away from the hole. It is less likely to be mapped back into the hole in the subsequent iterations and might even reduce the risk of this pixel getting stuck in a local minimum. The procedure, however, is rather expensive due to computing the cost eight times. This is especially severe in the context of parallel computation on the GPU, which causes many other threads to halt if even a single thread executes a different branch.

Random Neighbor  We can simplify the previously described approach by randomly choosing a neighbor and adding a random vector of length one. This way we only generate one candidate instead of considering many. This approach is much faster and turned out to produce results close or even better than the approach considering multiple candidates (see Section 5.8).

Convergence Criteria

The approach used for minimizing this cost function is local, and the cost function itself has many local minima. Therefore choosing when to stop the optimization is not trivial. The most straightforward solution is to stop after a fixed number of iterations. This allows for precise manual control and is simple to implement, but can easily under- or overfit.

A more advanced suggestion is to track the total cost \( \text{cost}(f) \) for all previous iterations and stop if the cost has not been reduced for a certain number of iterations. The mapping function generating the lowest cost can then be used as final mapping. This avoids under- and overfitting, but introduces a new parameter instead: For how many iterations is the total cost allowed to increase? It can also effectively improve the performance since only the smallest number of iterations necessary is performed.

3.2.3. Mapping Propagation to Next Mipmap Level

Once the optimization on a mipmap level \( L_{n-1} \) has stopped, we pass the resulting mapping function \( f_{n+1} \) to the next lower level \( L_n \). In this context, lower implies higher resolution by the definition of the mipmap levels. If the true border of \( T \) did not align with the pixel grid by chance, the target area on level \( L_n \) is enclosing the true hole more tightly than on level \( L_{n+1} \). We set the mapping to the identity for all source pixels according to the following definition:

\[
f_n(p') = p' \quad \forall p' \in S.
\]

We initialize the target pixels by the corresponding value of the last levels mapping. More precisely: \( f_n(p) = 2f_{n-1}(p//2) + (p_x \mod 2, p_y \mod 2) \) where \( // \) is the integer division and assuming \( p = (p_x, p_y) \in \{0, \ldots, \text{width}\} \times \{0, \ldots, \text{width}\} \) in integer coordinates. In case of using unified texture coordinates for \( \tilde{p} \in [0,1] \times [0,1] \) and bilinear interpolation this simplifies to \( f_n(\tilde{p}) = f_{n+1}(\tilde{p}) \).
3.2. Algorithm

3.2.4. Mapping Propagation to Next Frame

Due to the tracking of the physical object, we can compute the transformations describing the movement of the camera in between two frames. Therefore, we can use the approach discussed in Section 2.6.3. It consists of simple transformation of the mapping $\hat{f}_{n+1}^{\text{init}} = H^{-1}f_{n}H$ where $\hat{f}_{n+1}^{\text{init}}$ is the initialization of the mapping for the new frame, $f_{n}$ the final mapping function from the old frame and $H$ the homography describing the transformation of the object.

However, within the scope of this project, we did not implement this final part of the video inpainting pipeline. Instead, the current implementation treats every frame independently.
4

Implementation

Implementing the algorithm derived in the last chapter was motivated by two major goals: First, to evaluate whether video inpainting in real-time on mobile devices is feasible using the described approach. Second, to study the algorithm’s behavior and if necessary make changes. Studying the algorithm’s correctness necessarily precedes the performance evaluation. Therefore, we first made a prototype implementation in Python and once we sufficiently understood the algorithm’s behavior and results, the GPU implementation followed. However, since understanding the constraints on the implementation imposed by the GPU is essential, we first present the GPU implementation and discuss the Python prototype afterwards.

4.1. GPU Implementation

The requirements on the implementation of the AR inpainting pipeline presented in Section 1.3 imposed some restrictions on the choice of technology. We have emphasized on the advantage of leveraging the GPU for achieving the desired performance in the previous chapters. In this section, we will discuss how the GPU can be exploited best.

However, the GPU API landscape is by far not as unified as the one for CPUs, and hardware architecture, operation system and generation of the API are very diverse and highly specialized for specific applications. Even though general purpose GPU computation (GPGPU) has gained a lot more popularity in recent years, the corresponding technologies, such as OpenCL, CUDA, and C++ AMP, require either a specific hardware architecture, operations system or driver. Original intended for graphics processing only, GPU computing comes with a rich diversity of rendering APIs, e.g. Direct3D, OpenGL, Metal and Vulkan to name a few. They all support the execution of shader programs and have extended their API to allow dispatching the execution of shader programs independent of the rendering pipeline and added support for read-write data structures. To support devices based on Android, iOS and Windows devices at least OpenGLES,
4. Implementation

Metal and DirectX must be supported. These APIs all come with their own shading language used to define the programs executed on the GPU.

To avoid writing and maintaining three different code bases, cross-compilers exist that translate shader programs from one language to the other. The game engine Unity has incorporated such cross-compilers, along with a unified interface to the different rendering pipelines. These general purpose shader programs are called Compute Shader in Unity and are implemented using Cg which is syntactically mostly identical to HLSL. We use the Unity game engine in its 2017.1.3f version.

4.1.1. Coding for GPUs

With Unity, we found a framework to support deploying GPGPU code on different operation systems and devices. But, the algorithm needs to be implemented in HLSL and with standard shader programs. This results in some limitations we discuss in the following section.

Loop Unrolling  Loops get unrolled when compiled. As the cross-compiler is not efficient, a lot of temporary variables are created. These are limited, so loops must be kept short and have only a few iterations (we used at most loops with 24 iterations). This is particularly problematic for nested loops. Another problem with unrolling loops is that the number of iterations must be known at compile-time, branching out based on dynamic input is thus not possible.

Branching  When a shader program gets dispatched, a thread-group size is defined by the user. Thread-groups are a number of threads that get executed together. They can be forced to synchronize and share memory. A thread-group is executed in a single instruction multiple data fashion, which means that all threads are executing the same instruction simultaneously. This is in particular important for branching, as it creates multiple execution paths. Branching paths get serialized, and if even a single thread takes another path than the others, all threads will be forced to either execute that branch or be stalled. Thus avoiding unnecessary branching is crucial for exploiting the GPU’s potential. This makes the GPU unsuitable for algorithms that heavily rely on branching such as most search algorithms or which operate on tree data structures.

Input and Output  Traditionally shader programs take a set of read-only textures and parameters as input and produce no output other than the rendered image. Compute shaders, however, support read-write access textures and buffers that can be written to and read from both the CPU and GPU. In Unity, the read-write textures are called render textures and the buffers compute buffers. The number of render textures that can be used simultaneously is limited, and the exact number depends on the GPU. The same is true for the buffers that are additionally limited in size. Both the stride, the size of a single buffer element, and the number of elements are limited. The limit is dependent on the device as well. We were not able to use more than two render textures and two compute buffers at the same time without experiencing unexpected effects on the Shield tablet.
Data Structures  The data structures supported are limited to primitive types, such as floating- and fixed-point numbers and integers, as well as arrays of such. Furthermore, structured objects are supported and useful as buffer elements. Lists with dynamic lengths, dictionaries or classes are not supported. Textures either act as arrays and are indexed using integer coordinates or as textures in the traditional sense and accessed using floating number pairs between $(0, 0)$ and $(1, 1)$. When creating a texture, the data type of its elements is defined. They are mostly arrays of four values, corresponding to the four color channels RGB and alpha. The bit-size and number type, floating-, fixed-point or integer, can vary, but not all formats are supported on all devices. We used render textures with the format $RGBAFloat$ that consists of four floating point numbers per pixel with a precision of 32bit each.

Optimization  Typically a naive GPU implementation is not faster than an optimized CPU implementation. Optimizing GPU code can result in a dramatic performance boost and is unavoidable when aiming at high performance. The optimization of an implementation, however, is not within the scope and time available of this thesis. Therefore, the GPU code remained mostly unoptimized. Nevertheless, the implementation is fast enough for the evaluation of our supposition that video inpainting can be done in real-time on mobile devices.

4.1.2. Image Tracking

To identify the pixels to be inpainted, the object we want to remove is tracked. We place the physical object on a tracker image. This is an image with a distinctive pattern that can easily be recognized using image features. When this pattern is present on a camera frame, its position, scale, and orientation can be computed. This is sufficient to derive the relative position of the camera to the tracker image. The physical object is given as a 3D model and thus its exact shape and dimension are known as well. It is placed on the tracker image with a consistent orientation and its relative position to the camera is known as well.

We are using Vuforia [Vuf] in its version 6.2.10 for this purpose. Vuforia is available as a Unity extension and runs on all targeted platforms.

4.1.3. Scene Setup

Figure 4.1 depicts the scene setup discussed in the following. The Vuforia extension provides an AR camera asset representing the mobile device’s camera and an image target, representing the tracker image described in Section 4.1.2. The 3D model of the physical object is positioned atop the image target and the scene is set up for tracking the physical object. The camera stream used by Vuforia for identifying the tracker image is rendered to a background plane object that acts as a canvas in the background of the scene. Finally, there is a camera capturing the scene with the 3D model and the background plane.

Unfortunately, it was not possible to access the video stream rendered to the background plane directly on the Nvidia Shield K1 tablet. To the best of our knowledge, this is due to a bug in either Vuforia or the way the Shield tablet implements the Android camera API. The workaround
4. Implementation

**Figure 4.1**: Schematic of the Unity scene. The left camera represents the AR camera and background camera. The mask and camera frame get passed to the inpainting algorithm which processes the frame. The inpainted frame and the rendered object are then passed to the orthographic camera, which combines and renders them to the screen.

we eventually found is the following: An additional camera object, named background camera, captures the background plane V uforia uses to render the camera stream. This introduces an additional render pass, which increases the time needed for processing a frame.

Due to the tracking, the 3D model of the object always correctly aligned with the image of its physical counterpart on the camera frame. The mask texture that defines the hole is created by rendering the 3D model without shading in a uniform color. The background remains black indicating the source area $S$ while any pixel with color is part of the target area $T$.

The mask and camera frame are passed to the inpainting algorithm as input. The inpainting consists of various compute shader programs. They get the input data and dispatch calls from a C# script that gets triggered every time Unity starts rendering a new frame. Eventually, the frame with the inpainted image and a texture containing the rendered 3D model are passed to a third Unity camera, labeled as orthographic camera on Figure 4.1. The orthographic camera overlays the two textures and renders them to a quad, render quad on Figure 4.1, that is displayed on the screen using an orthographic projection.

### 4.1.4. Mapping Function as Texture

The domain of the mapping function $f : T \rightarrow S$ is the set of the hole or target pixels $p \in T$. $T$, however, depends on which mipmap level is considered. Thus, the representation of $f$ also depends on the mipmap level. Since the mapping function we are optimizing for is defined on
4.1. GPU Implementation

pixels, it can be represented as a matrix $F: F_{ij} = f([x_i, y_j])$. This matrix is stored as a render texture with four floating point values per pixel. The first two contain the mapping $F_{ij}$, they are the $x$- and $y$- components of this texture’s pixel. The third, the $z$-component, is 1 if the pixel is a target pixel and 0 if it is a source pixel. Finally, the $w$-component of the mapping texture’s pixels is the fourth entry which is 1 if the mapping for this pixel is valid and 0 otherwise. The mapping of a pixel is invalid, if it is not initialized yet, or lays outside the image boundaries. Two versions of the mapping are used: The old mapping as read-only texture containing the mapping from the previous iteration and the new mapping as read-write texture, that takes the newly computed mapping coordinates. Between two iterations, they are swapped and the next iteration takes the former new mapping as input.

4.1.5. Initialization

The first step of the algorithm is initializing the mapping based on the provided mask and camera frame. This requires iterating all target pixels and searching the source pixels to find the one synthesizing the lowest cost. Merely iterating all possible pairs of source and target pixels using a loop is not possible using compute shaders due to loop unrolling and resulting long execution times. Instead, one thread is assigned for each source and target pixel pair. All threads processing the same target pixel are grouped and can thus use shared memory. They compute the cost for their assigned pixel pair and store it into an array that lives in the shared memory. Following the computation of the cost, the threads are synchronized and the source pixel with minimum cost is identified. Finding the minimum of an array can be done by a parallel reduction.

Before the initialization shader program can be dispatched on the GPU, the input data has to be loaded into the textures and buffers. The camera frame is retrieved as a texture from the background camera and can simply be bound to the shader. All the input parameters are stored into a structured buffer and a second buffer is used to flag pixels as initialized. Using the mask texture, the coordinates of source and target pixels are identified on the CPU and stored into a texture. Storing them into a buffer would be more convenient, but at most two buffers can be bound to a shader program in order to stay compliant with the OpenGL default limits.

For both, the old and the new mapping, the source pixels get set to the identity mapping $f(p') = p' \forall p' \in S$. The $z$- and $y$-components get initialized as described in Section 4.1.4. Then, the shader program for initializing the target pixels is dispatched. Only target pixels with a sufficient number of valid neighbors can be processed. After the first batch of pixels has been processed, the mapping textures are swapped and the process is repeated until all target pixels are initialized.

4.1.6. Neighborhoods

Both, the color and spatial cost terms operate on a neighborhood around the processed pixel $p$. The size and shape of the neighborhoods require careful evaluation. Figure 4.2 shows the three kinds of neighborhoods used.

The neighborhood used in the spatial cost term must be large enough to enforce a patch-like
4. Implementation

structure. Using the 8-neighborhood instead of the four principal neighbors leads to an improvement. Increasing the size even more did not clearly improve the result, as discussed in Section 5.9.

For the color cost term, a neighborhood too small will not capture the structures. However, computing the color cost gradient includes computing the color cost four times. Using any neighborhood larger than the 12-neighborhood exceeds the maximum number of temporary variables, because of the loops unrolling. Thus using the 12-neighborhood was the best option, though we assume that a larger neighborhood might be able to capture high-frequency structures better.

4.1.7. Optimization

The optimization takes the old mapping containing the initialized mapping and writes the newly computed mapping coordinates into the new mapping texture. To check whether a mapping of a pixel points into the hole, the mask texture is used.

Each target pixel gets assigned a thread that first computes the gradient of the cost function, and then checks if the new mapping points into the hole. If this is the case, the random neighbor procedure is applied. Finally, the mapping is clamped to the edges of the image borders if it lays outside, and is written into the new mapping render texture.

This is repeated for a fixed number of times and between each iteration, the two mapping textures are swapped.

4.1.8. Mapping Propagation

When moving to a higher resolution level in the image pyramid, the mapping from the previous level is used as initialization. The implementation is not strictly following the discrete definition from Section 3.2.3, but uses a hybrid approach of both definitions, the one using discrete coordinates and the one using unified texture coordinates. This is due to the fact that the mapping
is stored as a discrete texture and can be made continuous using bilinear interpolation. When implementing the mapping propagation in shader code, was assumed that using bilinear interpolation on the mapping texture is the best approach. However, the interpolated values can only be accessed using unified texture coordinates. Thus, the discrete coordinates of the target pixels have to be transformed into unified texture coordinates, which are then used for looking up the interpolated value. This is done for all target pixel on the new mipmap level. The resulting mapping is then the initialization to the optimization.

### 4.1.9. Mapping Texture Access

The idea of using the gradient descent method initially motivated to consider the mapping as a continuous function. The data stored in the texture, however, is inherently discrete and is made continuous by using bilinear interpolation. This introduces two different coordinate systems for the mapping texture:

1. Using integer pixel indices in \( \{0, \ldots, w-1\} \times \{0, \ldots, h-1\} \), where \( w \) is the texture width and \( h \) its height. This is identical to accessing a two dimensional array.

2. Using unified texture coordinates with values in \([0, 1] \times [0, 1]\). These are also called uv-coordinates with \( u \) and \( v \) instead of \( x \) and \( y \). Additionally, the mapping texture is padded with a margin of one pixel on all sides. When trying to access a pixel outside the valid texture coordinates, it is clamped to the outermost row or column. These rows and columns have a \( w \)-component that is 0, marking them as invalid mappings. This makes additional boundary checks unnecessary and avoids unnecessary branching on the GPU. This padding shifts all coordinates by one pixel.

When using uv-coordinates, the coordinate \((0, 0)\) is the bottom left corner of the bottom left pixel\(^1\). We want to access the center of the pixel because of the interpolation though. Sampling the center of the pixel returns the same value as when accessing the texture as an array using integer indices. Therefore, the uv-coordinates must be shifted by half a pixel in both dimensions.

All the above should be transparent to the algorithm which operates on integer coordinates with \((0, 0)\) as the first valid pixel in the bottom corner.

This results in six different coordinate systems for accessing and sampling the mapping texture.

1. Padded using integer coordinates \( (x_o, y_o) \)
2. Padded using uv-coordinates \( (X_o, Y_o) \)
3. Not padded using integer coordinates \( (x_i, y_i) \)
4. Not padded using uv-coordinates \( (X_i, Y_i) \)
5. Not padded and corrected for uv-coordinates, in integer coordinates \( (\tilde{x}_i, \tilde{y}_i) \)
6. Not padded and corrected for uv-coordinates, in uv-coordinates \( (\tilde{X}_i, \tilde{Y}_i) \)

---

\(^1\)This is following the OpenGL definition, DirectX uses a different convention, but Unity hides these differences to the programmer.
4. Implementation

The first is used when accessing the mapping as an array and the second when using uv-coordinates. Any interpolated texture access has to be performed using the second coordinate systems, so the others have to be transformed first. The algorithm is using the third kind of coordinates and needs an interpolated view on the mapping. The sixth coordinate system is used when sampling the camera frame and mask.

Transformations

Due to the complexity introduced by the different views on the mapping textures, we discuss the transformation from one system into another in this section. This is done for the $x$-component only, the $y$-component works analog, using the height instead of the width.

For each coordinate system, the width of the texture is denoted with $w$ and the same subscripts as the indices used for this system. For example $w_o$ when using padded integer coordinates $(x_o, y_o)$. The uppercase notation $X$ denotes uv-coordinates and lowercase $x$ the integer coordinates.

In general the equation $X = x/w$ hold for all coordinate systems. Additionally, the relation between the systems is given as:

$$x_o = x_i - 1 = \tilde{x}_i - 1.$$  \hfill (4.1)

This yields

$$X_o = \frac{x_o}{w_o} = \frac{x_i + 1.5}{w_o},$$ \hfill (4.2)

which is used for accessing the interpolated mapping values in the initialization and the mapping refinement steps.

When rendering the final frame, the $(\tilde{X}_i, \tilde{Y}_i)$ coordinates are given. Using the above equation, they are transformed for the mapping texture lookup in the following manner:

$$X_o = \frac{\tilde{X}_i \tilde{w}_i + 1.5}{w_o}.$$ \hfill (4.3)

4.2. Python Prototype

The prototype of the algorithm was implemented in Python version 3.6 using numpy [ea] for extended array operation support and Cython [RB] for performance improvement. The implementation simulates the constraints from the GPU. Since it does not come with the benefits of a GPU it is much slower. Processing a single frame at a resolution of $512 \times 512$ pixels takes 4.5h, whereas the GPU implementation takes 352.62ms. Working on this prototype allowed us to quickly try new ideas and understand their effects more easily thanks to good debugging tools and the ability to easily plot results. The prototype does not support any of the augmented reality components.

The Python implementation has some features, which are not implemented in the GPU version. It uses the convergence criterion based on the total error described in Section 3.2.2 which helps to reduce the computational cost and to avoid over-fitting.
4.2. Python Prototype

Some parameter values are depending on the mipmap level, most relevant the $\alpha$-weight and a parameter to control the step size of the gradient descent minimization. Since this introduces more parameters that have to be set manually, we did not incorporate this in the GPU implementation.

Furthermore, the selection of the mipmap level for the initialization is dependent on the number of source and target pixels. First, the level with at least three source and one target pixel $L_b$ was identified and then the $L_{b+2}$ level is selected for the initialization. This most often leads to an initial resolution of $8 \times 8$ pixels. However, experiments have shown that $16 \times 16$ pixels works best in most cases. If the size of the hole does not vary drastically, this provides the simpler initial condition and is used for the GPU implementation.

Initially, we wanted to not only refine the mapping level by level but also be able to go back to a lower resolution if the mapping became unstable. This would preserve the high-frequency texture structures while readjusting the entire patches to the low-frequency details. Therefore, the mapping data structure is designed hierarchically as a pyramid $M$ of arrays $M_m$ representing the mapping for the $m$-th mipmap level of the image. Every layer of this pyramid only stores the adjustments made during the optimization of its mipmap level only. Thus, when reading a mapping coordinate for the $m$-th level from this pyramid $M_m$, all layers from $m$ to the level of the initialization $M_l$ have to be considered. The mapping of pixel $(y, x)$ at mipmap level $L_m$ is then given by $M_{\text{tot}}(y, x) = \sum_{i=m}^{l} 2^{i-m} M_i(y/2^{i-m}, x/2^{i-m})$ where $//$ is the integer division. Dividing the coordinates by two at each step up the pyramid is necessary because the resolution gets smaller and the pixel size larger with each level. Multiplying the result by $2^{\text{steps up the pyramid}}$ is necessary for the same reason.
Results

We will present and discuss the results produced by our implementation of the proposed algorithm. First we present an overview on the results we produced using our algorithm, and introduce the method used to generate the images. We then provide a performance evaluation of the implementation and discuss the computational cost of the different steps in the inpainting pipeline. Then we discuss the impact of the $\alpha$-weight parameter, introduced in Equation (2.1), and the mapping interpolation. They are the two most important parameters of our algorithm and therefore deserve a detailed evaluation. We also discuss how changing individual parameters, such as $\tau$ for clamping the spatial cost gradient, the normalization parameter of the color cost, or the size of the spatial cost neighborhood, affects the results. In the remaining part of the chapter, we present how we chose the best procedure to deal with pixels that get mapped into the hole, and show two limitations of our work. First, we discuss the effects of undersampling the image during the initialization. Second, we show that our results depend on the image orientation.

5.1. Result Overview

Our algorithm produces very good results in many different scenarios. We can replace small objects (see Figures 5.1g and 5.1i), or entire buildings (see Figure 5.1h) with a credible reconstruction of the background. Holes as large as 10% of the image size are processed within 1.3 seconds for a resolution of $512 \times 512$ pixels on desktop computer (see Section 5.4 for the specifications). The high-frequency details of the texture are preserved and structures continued into the hole in a natural manner, and the color transitions are smooth.
5. Results

![Images of bunny, mill, and zucchini images with masks applied and results produced using algorithm.](images)

**Figure 5.1.** Results produced using our algorithm. The textures are successfully recovered, structures continued, and the color transition smooth.

### 5.2. Method

All images were captured using RenderDoc a graphics debugger that serializes graphics API calls and allows capturing individual frames, textures, and buffers. This guarantees that all images are exactly as presented by our implementation without adding or hiding any interpolation or distortions.

The images used in this chapter are labeled with a reference name, e.g. `lausanne2`, and if multiple masks for a single image exist, they are enumerated starting with zero. All images are depicted in Appendix A with references to the respective creators.

Our implementation either runs in the AR-mode that uses image tracking, or takes a single image along with a mask as input. For most of the results in this chapter, we use single images and manually created masks as input. The results, for which we used the AR-mode, are explicitly labeled as such.

### 5.3. Parameters

For all results, the parameters used are summarized in a table. If the value of a parameter is not explicitly mentioned in the table, or given as default values in Table 5.1, it correspond to the
5.4. Performance

Table 5.1.: Default parameters.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Init width</th>
<th>Color gradient step size</th>
<th>Spatial gradient step size</th>
<th>$\omega_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>16px</td>
<td>1</td>
<td>0.25</td>
<td>4</td>
</tr>
</tbody>
</table>

value mentioned in chapter Chapter 4. The parameters always indicated are:

- $\alpha$: The weight used to balance the color and spatial cost.
- Init width: The width in pixels of the mipmap level used to initialize the mapping.
- Levels: The number of mipmap levels used for the optimization, starting with the one used for initialization.
- Iterations: The number of iterations of the optimization performed per mipmap level.
- Interpolation: Whether interpolation of the mapping was used or not.

5.4. Performance

The performance of the algorithm is crucial if it should run in real-time. However, we never optimized the implementation for performance, so an in detail analysis is not meaningful at this point. The following section aims at giving an understanding of how much of a frames time is spent on which stage of the algorithm.

The measured values are:

- $\frac{\#T}{\#I}$ where $\#T$ is the number of target pixels $p \in T$ and $\#I$ the total number of pixel. This is to measure the hole size.
- $Total$: The total time in $ms$ spent on processing a single frame from creating the image to finally rendering a frame.
- $Initialization$: The time in $ms$ used for the initialization of the mapping.
- $Refinement$: The time in $ms$ spent on the optimization. The $Total$ of the refinement includes all optimization levels and iterations, while $Level\ n$ is the duration of optimizing the $n$th mipmap level only.

All measurements are made in the CPU code, which introduces a small overhead. The precision of the measurements is 1 $ms$. Approximately 100 frames were used for each measurement, excluding a warm-up time of 10 frames. All measurements, along with all other results in this chapter, were created on a desktop computer with an Intel i7 4790K @4.00GHz, Nvidia GeForce GTX 980 and 16GB DDR3 RAM.

Table 5.2 contains the resulting measurements for six setups. The $AR$-mode can only process four levels due to texture memory constraints. Therefore, the lausanne2 scene has been considered twice. Once processing six levels from 16 by 16 pixels up to $512 \times 512$ pixels, and only using four levels up to a resolution of $128 \times 128$ pixels. The values presented in Table 5.2 and
in the \textit{Total} column of Table 5.3 are the average, minimum, maximum and standard deviation using the following format: average (min/max/standard deviation). Table 5.2 breaks down the optimization into the individual mipmap levels in the order of processing. The first level has a width of 16 pixels, the second twice the width and so on, up to level 6 with 512 pixels width.

When comparing the results of \textit{lausanne2}, \textit{4 levels} and \textit{AR}, a big difference in all values is observable. For processing approximately the same number of pixels, the \textit{AR}-mode takes more than 5 times longer (236.10ms vs. 45.64ms) for processing a single frame. The initialization even takes more than 20 times longer. Given that the GPU code and most parts of the CPU code are the same for both setups, the remaining components to attribute this difference to, is the image tracking introduced in Section 4.1.2 and the Unity scene setup discussed in Section 4.1.3. The time to create the background image and mask in the \textit{AR}-mode is negligible, at most 1ms and the tracking is performed on the CPU, should thus not affect the initialization much. Whatever causes this performance loss, the fact that only the \textit{AR}-mode suffers from it, strongly implies no connection to the algorithm and its GPU implementation. It can, therefore, be considered as an implementation issue.

The other results show that the \textit{Total} and \textit{Refinement} time depend on $\#T \div \#I$. More target pixels require more time for processing. The \textit{Initialization} time however, is less affected by this. Its complexity is $O(\#T \cdot \#S)$ and a larger number of target pixels results in a smaller number of source pixels. Furthermore, the precision of the measurements is not high enough to accurately capture the initialization.

The computation time for levels with higher resolution is clearly increases superlinearly with respect to the mipmap level. The number of pixels increases by a factor of four when moving to the next level due to the increased resolution. However, at the same time the mask is enclosing the true target area more and more tightly, resulting in a smaller total area to be inpainted.

The standard deviations of \textit{Total} and \textit{Optimization} imply a distribution that is fairly narrow, however, the assumption of a normal distribution has never been verified. Since there is minimal branching in the GPU code that could drastically affect the computational cost of a single frame, it is a reasonable assumption, though.

Finally, the time used for disposing of textures and buffer objects after rendering the frame is close to 1ms.

\textbf{Mobile Device} The implementation for mobile devices is limited to the \textit{AR}-mode, for which the measurements reveal some anomalies, as discussed above. So only a simple performance test was made, during which the frame times, corresponding to the \textit{Total} in Table 5.2, were read from the screen. The first measurement is without optimization. It includes the tracking, creation of the mask and the image, and the initialization. It can thus not be compared to the measurements of the \textit{Initialization} value discussed before. The following measurements include one more mipmap level of optimization at a time. The measurement for 2 levels therefore include the initialization and the optimization on the first two mipmap levels. This is not the same as in Table 5.3 where the times correspond to the processing of a single level exclusively. The results are in Table 5.4 and the comparison between \textit{No optimization} (400ms) and \textit{1 level} (450ms) clearly show that either the initialization or the preparing steps before the initialization require most of the time. Previous results have shown that the initialization is fast, which leaves
the preparing steps and the tracking as the suspects for the reduced performance. The device used is a Nvidia Shield K1.

**Conclusion** These numbers can be compared to the Python prototype using the building image with mask 0. The Python implementation took 4h34min to process a single frame up to the resolution of $512 \times 512$ pixels. The GPU implementation on the desktop computer requires 352.62ms per frame. For this example, the GPU implementation is almost 50'000 faster.

The Python implementation requires approximately 14 minutes for processing the building image with mask 0 to a resolution of $128 \times 128$ pixels. The Shield tablet requires 236.10ms for processing a frame with a slightly larger target area up to a resolution of 128 pixels width. This still yields a processing time more than 3000 times faster on the tablet computer over the Python implementation. We do not want to put much weight on this comparison, it is based on a single example and compares two implementations made for a very different purpose. Nevertheless, these numbers help to get a grasp on the dimension of the performance difference, which can be expected in comparable scenarios.

<table>
<thead>
<tr>
<th>Scene name</th>
<th>#/T</th>
<th>Total [ms]</th>
<th>Initialization [ms]</th>
<th>Refinement [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lausanne2, 6 levels</td>
<td>0.032</td>
<td>364.44 (354/371/4.29)</td>
<td>0.62 (&lt;1/2/0.54)</td>
<td>362.23 (352/370/4.28)</td>
</tr>
<tr>
<td>lausanne2, 4 levels</td>
<td>0.037</td>
<td>45.64 (37/104/13.35)</td>
<td>0.61 (&lt;1/7/0.83)</td>
<td>43.71 (35/101/12.95)</td>
</tr>
<tr>
<td>building, mask 0</td>
<td>0.023</td>
<td>352.62 (345/364/3.69)</td>
<td>0.45 (&lt;1/1/0.50)</td>
<td>350.45 (343/362/3.70)</td>
</tr>
<tr>
<td>building, mask 1</td>
<td>0.040</td>
<td>397.20 (393/404/3.27)</td>
<td>0.63 (&lt;1/1/0.49)</td>
<td>395.11 (391/402/3.22)</td>
</tr>
<tr>
<td>building, mask 2</td>
<td>0.056</td>
<td>450.17 (442/458/3.88)</td>
<td>0.77 (&lt;1/1/0.43)</td>
<td>447.94 (440/456/3.92)</td>
</tr>
<tr>
<td>AR</td>
<td>0.034</td>
<td>236.10 (177/269/32.72)</td>
<td>21.22 (16/30/3.28)</td>
<td>173.94 (128/199/24.33)</td>
</tr>
</tbody>
</table>
5. Results

**Table 5.3.:** Measurements per mipmap level. The Total column’s values are structured as <average (min/max/standard deviation)>. All measurements are in ms. The parameters used are in Table 5.5.

<table>
<thead>
<tr>
<th>Scene name</th>
<th>Total [ms]</th>
<th>Level [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. 2. 3. 4. 5. 6.</td>
<td></td>
</tr>
<tr>
<td>lausanne2, 6 levels</td>
<td>362.23 (352/370/4.28)</td>
<td>3.15 4.15 7.00 22.08 67.95 254.62</td>
</tr>
<tr>
<td>lausanne2, 4 levels</td>
<td>43.71 (35/101/12.95)</td>
<td>3.06 4.64 8.60 25.24</td>
</tr>
<tr>
<td>building, mask 0</td>
<td>350.45 (343/362/3.70)</td>
<td>2.79 3.57 7.38 17.77 70.06 245.62</td>
</tr>
<tr>
<td>building, mask 1</td>
<td>395.11 (391/402/3.22)</td>
<td>2.26 3.69 6.80 26.34 75.49 277.03</td>
</tr>
<tr>
<td>building, mask 2</td>
<td>447.94 (440/456/3.92)</td>
<td>2.17 4.00 13.34 23.63 85.00 316.54</td>
</tr>
<tr>
<td>AR</td>
<td>173.94 (128/199/24.33)</td>
<td>12.20 49.10 50.10 60.48</td>
</tr>
</tbody>
</table>

**Table 5.4.:** Performance evaluation on a Nvidia Shield tablet. All values are in ms. The parameters used are in Table 5.5.

<table>
<thead>
<tr>
<th>#T #I ratio</th>
<th>No optimization</th>
<th>1 level</th>
<th>2 levels</th>
<th>3 levels</th>
<th>4 levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.034</td>
<td>400</td>
<td>450ms</td>
<td>750</td>
<td>1000ms</td>
<td>1700ms</td>
</tr>
</tbody>
</table>

**Table 5.5.:** Parameters for Section 5.4.

<table>
<thead>
<tr>
<th>α</th>
<th>Init width</th>
<th>Levels</th>
<th>Iterations</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>16px</td>
<td>6 (if not indicated differently)</td>
<td>30</td>
<td>off</td>
</tr>
</tbody>
</table>

5.5. α-Weight and Interpolation

The most sensitive parameter is the α-weight that balances the color and spatial cost terms. If not appropriately chosen, one term dominates the optimization. If the color cost dominates, the texture is inconsistent, the mapping not sufficiently smooth and the low-resolution pixel structure remains preserved. This results in pixels being mapped to blocks rather than seamlessly fitting patches of texture. If, on the other hand, the spatial term dominates, the border between source and target area is clearly visible due to not adapting to the surrounding color sufficiently. Structures look more random and color gradients are not smooth.

The second question is whether bilinear interpolation of the mapping leads to a better result. However, experience has shown that both questions are tightly coupled. The same α-weight can lead to a good result when not interpolating and to a bad result when interpolating.
In Table 5.6 the results for $\alpha \in \{0.0, 0.1, \ldots, 1.0\}$ are depicted, both with and without interpolation of the mapping. For the lausanne2 image on the left, $\alpha$ between 0.5 and 0.8 produces the best results without interpolation. Turning on the interpolation results in a smoother inpainted area, but adapts to the surrounding texture to a lesser degree. Using bilinear interpolation the sensitivity to $\alpha$ is less significant.

The building image shown in the right two columns of Table 5.6 produces the best results for $\alpha \in [0.7, 0.9]$. Any lower values fail to continue the structure of the roof and degenerate completely for values below 0.2. $\alpha = 1.0$ continues the roof’s structure better than lower $\alpha$ values, but the texture of the sky shows a clear block structure. This is not surprising, given that the spatial cost term has been ignored completely, making the mapping keep the brick-like structure of the low-resolution optimization steps. The interpolation has a much smaller impact than in the lausanne2 example, slightly distorting the texture of the sky for high $\alpha$ values.

In conclusion, it is hard to pick a value for $\alpha$ that performs good on all examples, but in general $\alpha \in [0.6, 0.9]$ is a good range. So far there is no way of automatically predicting or deriving a good $\alpha$ since the quality of the result remains to be assessed by a human observer.

Structures and uniform areas mostly profit from mapping interpolation, whereas textures get blurry. There is also an inherent risk of introducing additional artifacts by interpolating the mapping. This can be seen in Figure 5.2. Two neighboring pixels ($p$ and $p'$ in Figure 5.2d) are mapped far apart to areas of similar color ($f(p)$ and $f(p')$ are both in a black area), but with an area of different color in between (white). When sampling the mapping in between the center of $p$ and $p'$ we get a mapping that is between $f(p)$ and $f(p')$. For example, for the pixel $\frac{p+p'}{2}$ this is $f\left(\frac{p+p'}{2}\right) = \frac{f(p)+f(p')}{2}$. In the example of Figure 5.2 $\frac{f(p)+f(p')}{2}$ is in the white area. In Figure 5.2c the artifacts resulting from this interpolation are well visible as white structures in the lower black area.

In the best case, using mapping interpolation slightly improves structures and smooths uniform areas, in the worst case, it introduces disturbing artifacts. We do not interpolate by default.
5. Results

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>No interpolation</th>
<th>Bilinear interpolation</th>
<th>No interpolation</th>
<th>Bilinear interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td><img src="image1" alt="Original Image" /></td>
<td><img src="image2" alt="Masked Image" /></td>
<td><img src="image3" alt="Original Image" /></td>
<td><img src="image4" alt="Masked Image" /></td>
</tr>
<tr>
<td>0.1</td>
<td><img src="image5" alt="Original Image" /></td>
<td><img src="image6" alt="Masked Image" /></td>
<td><img src="image7" alt="Original Image" /></td>
<td><img src="image8" alt="Masked Image" /></td>
</tr>
<tr>
<td>0.2</td>
<td><img src="image9" alt="Original Image" /></td>
<td><img src="image10" alt="Masked Image" /></td>
<td><img src="image11" alt="Original Image" /></td>
<td><img src="image12" alt="Masked Image" /></td>
</tr>
<tr>
<td>0.3</td>
<td><img src="image13" alt="Original Image" /></td>
<td><img src="image14" alt="Masked Image" /></td>
<td><img src="image15" alt="Original Image" /></td>
<td><img src="image16" alt="Masked Image" /></td>
</tr>
</tbody>
</table>
5.5. $\alpha$-Weight and Interpolation

- 0.4
- 0.5
- 0.6
- 0.7
- 0.8
- 0.9
5. Results

Table 5.7: Parameters for Table 5.6.

<table>
<thead>
<tr>
<th>α</th>
<th>Init width</th>
<th>Levels</th>
<th>Iterations</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>varying</td>
<td>16px</td>
<td>6</td>
<td>30</td>
<td>varying</td>
</tr>
</tbody>
</table>

Table 5.6: Comparing different α values. The left two column show the lausanne2 image. The interpolation of the mapping is turned off for the first column and set to bilinear filtering for the second column. The same on the third and fourth column for the building image using mask 2. The parameters used are in Table 5.7.

5.6. τ - Clamping of Spatial Gradient

In Equation (3.10), we use the clamped $L_2$-norm for the robust distance function $d_s$ with τ as clamping threshold. Note that the maximum possible value for the spatial cost term is independent of τ, as it gets normalized as discussed in Section 3.1.2.

The impact of τ is depicted in Figure 5.3. τ = 2 results in a mapping with block structures, indicating that the spatial cost term is not strong enough to smoothen the mapping across sharp jumps. The visualization of the cost per pixel shows the same: A high color cost dominating the spatial cost. τ = 32 is the opposite: It creates a maximal spatial cost at the border of patch-like structures. Finally τ = 4 results in a smooth mapping and a balanced contribution of color and spatial cost. Though the cost and mapping show an apparent sensitivity to the value of τ, the final inpainted result is not significantly affected. Experiments with other images have shown the same behavior. Based on observations it seems reasonable to set τ to at least 4 and no larger than twice the neighborhood size used.

Table 5.8: Parameters for Figure 5.3.

<table>
<thead>
<tr>
<th>α</th>
<th>Init width</th>
<th>Levels</th>
<th>Iterations</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>16px</td>
<td>6</td>
<td>30</td>
<td>off</td>
</tr>
</tbody>
</table>
5.7. Color Cost Normalization

In Section 3.1.2, we introduced a color cost normalization. Figure 5.4b shows the whiteboard scene without normalization, Figure 5.4c using the standard deviation for normalization and Figure 5.4d using twice the standard deviation. Using the standard deviation for normalization slightly improves the result, while using twice the standard deviation affects the result negatively. Figures 5.4e and 5.4g show the visualization of the cost per pixel. Red is the total cost, green the color cost and blue the spatial cost.

Without normalization, the spatial cost dominates, and with the standard deviation as the normalizer, the color cost is higher and has a larger impact on the total cost. We desire to keep the color and spatial costs on a similar level. This means that neither of both terms dominates the optimization. When we further increasing the color cost and use twice times the standard deviation for the normalization, the color cost term decreases. This indicates that the color term dominates the optimization to the point that the spatial cost was not strong enough to act as a counter force.

Also, note how the purple lines (the color error) resemble a grid. The spatial error is highest,
5. Results

where neighboring pixels are not mapped to the same patch. This typically happens when the patch-structure from a low-resolution mipmap level get preserved. This results in sharp lines that indicate the separation of two patches, which are far apart.

| Table 5.9.: Parameters for Section 5.7. |
|---|---|---|---|
| α | Init width | Levels | Iterations | Interpolation |
| 0.9 | 16px | 5 | 25 | off |

5.8. Mapping Back into the Hole

In Section 3.2.2 we discuss four approaches to deal with pixels that get mapped into the hole. We had performed a detailed study of the once labeled *Force, Random neighbor* and *Neighbor with smallest error* with the Python implementation. In this section we discuss the results of this study. Note that the Python implementation uses the second convergence criterion discussed in Section 3.2.2, so the number of iteration is varying.

The results for the *lausanne1* image are in Figure 5.5 and for the *building* image in Figure 5.6. The red pixels in some results indicate that the procedure had failed and this pixel was mapped into the hole nevertheless.

In Figure 5.5 *Random Neighbor* and *Neighbor with Smallest Error* outperform *Force* in terms of inpainting quality. *Random Neighbor* is even slightly better around the top of the tower compared to both others. In Figure 5.6 three different masks were used on the same image, showing an inconsistent outcome. While *Mask 0* (Figures 5.6e to 5.6g) shows good and very similar results for *Force* and *Neighbor with Smallest Error*, *Random Neighbor* performs much worse. The opposite is true for the results of *Mask 1* (Figures 5.6h to 5.6j) where *Random Neighbor* clearly outperforming the other two. And finally the results for *Mask 2* (Figures 5.6k to 5.6m) doubtlessly favor the *Force* approach.

We decided in favor of the *Random Neighbor* procedure. It is the fastest any results in a similar quality as the other approaches on average.

| Table 5.10.: Parameters for Figures 5.5 and 5.6. |
|---|---|---|---|
| α | Init width | Levels | Iterations | Interpolation |
| 0.9 | 8px | 8 | 15-30 | off |

5.9. Spatial Cost Neighborhood

We discussed the importance of the size of the neighborhood in Section 4.1.6.
5.10. Undersampling and Rotation

Figure 5.7 shows the result of using the 8-neighborhood and the 12-neighborhood on three examples. The choice of the neighborhood does affect the result. However, it is hard to tell which is better, both display some weaknesses. The 12-neighborhood seems to preserve the structure of the roof better in Figure 5.7f, but introduces high-frequency artifacts. In Figure 5.7d the tip of the removed clock tower is filled with a texture strongly resembling the surrounding mountains when using the 24-neighborhood. Using the 8-neighborhood results in a more uniform inpainting, as seen in Figure 5.7a.

We used the 24-neighborhood for all results in this chapter because the increased computational costs are negligible and it is the more robust option.

<table>
<thead>
<tr>
<th>Table 5.11.: Parameters for Figure 5.7.</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0.8</td>
</tr>
</tbody>
</table>

5.10. Undersampling and Rotation

As mentioned in Section 3.2.1, the exhaustive search performed during the initialization, introduces an error. We do not sample the image at twice its maximum frequency, as required by the Nyquist–Shannon sampling theorem. As a result, good matches are not considered, because they do not align with the coarser grid of 1 pixel width. Additionally the color cost term is not rotation invariant, further excluding good matches, such as $f'(p)$ in Figure 5.8.

In Figure 5.8 these limitations lead to a bad initial mapping for $p$, the color border brown/black at the top edge of the image is the best approximation to the corner of the paper to be inpainted.

When adding another white piece of paper (second row of Figure 5.8), a better candidate is available, and the pixel will map there. The mapping of $p$ and all pixels which depend on $p$ as a neighbor during the initialization can be significantly affected as a result.

Note that when computing the color cost for a pixel $p$, the color of $p$ itself is not considered, but only its neighboring pixels’ colors. By introducing details, such as the pen, and not sampling sufficiently dense, the resulting inpainting can show unpleasant high-frequency artifact.

<table>
<thead>
<tr>
<th>Table 5.12.: Parameters for Figure 5.8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

5.11. Shadows

We discussed and showed how we can track and remove an object from an image. Depending on the light, however, objects cast a shadow visible on the image as well. Without knowing the
5. Results

position of the light source, the shadow cannot be tracked and therefore not removed either.

Figure 5.9 shows such an example. The shadow, outlined in Figure 5.9c, remains visible and affects the illusion despite the successful removal of the bunny sculpture from the image. This is a limitation of our work.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\alpha & \text{Init width} & \text{Levels} & \text{Iterations} & \text{Interpolation} \\
\hline
0.8 & 16px & 6 & 30 & off \\
\hline
\end{array}
\]

Table 5.13.: Parameters for Figure 5.10 and Figure 5.9

5.12. Rotation dependency

While testing the AR-mode, we observed an anomalous behavior from time to time. In specific scenarios, the inpainting seems to consider the image content at the bottom of the hole more than the other image areas. This effect can easily be verified by mirroring and rotating a frame that captures this behavior. In the ideal case, all rotations should produce the same result. However, this is not the case. Figure 5.10 shows the resulting images of various rotations of a frame taken using the AR-mode. The results are different and mostly reproduce colors and structures similar to the image content below the source area.

We first understood this effect when already writing this report, so a deeper analysis of the underlying problem was not possible anymore. The reason it could remain undiscovered for so long is that in many cases the effect is not so obvious and that we have intensely reviewed the GPU code on various occasions.

We observe the same effect with the Python implementation. This leaves two possible explanation: First, both implementations are suffering from the same bug or from bugs causing a very similar effect. Though unlikely, this is not impossible. It was the same person with the same conception of the algorithm that created both the GPU and the Python implementation. Second, this effect is due to a flaw in the design of the algorithm.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\alpha & \text{Init width} & \text{Levels} & \text{Iterations} & \text{Interpolation} \\
\hline
0.8 & 16px & 6 & 30 & off \\
\hline
\end{array}
\]

Table 5.14.: Parameters for Figure 5.10
5.12. Rotation dependency

Figure 5.3: Comparison of different values for the threshold \( \tau \) used for the robust distance function in the spatial cost term. First row: Resulting inpainting; Second row: Cost visualized per pixel (red: total cost, green: color cost, blue: spatial cost); Third row: Resulting mapping. The building scene with mask 0 was used and the images cropped. The parameters used are in Table 5.8.
5. Results

**Figure 5.4.** Comparison of different color cost normalizations for the whiteboard image. The first row shows the resulting images with different normalization factors. The second row visualizes the cost per pixel (red: total cost, green: color cost, blue: spatial cost). The parameters used are in Table 5.9.

**Figure 5.5.** Comparison of three approaches to avoid mapping pixels into the target area. Image used: lausanne1. The parameters used are in Table 5.10.
5.12. Rotation dependency

Figure 5.6.: Comparison of three approaches to avoid mapping pixels into the target area. Image used: building. The parameters used are in Table 5.10.
5. Results

Figure 5.7.: Comparison of different neighborhoods used in the spatial cost term. The parameters used are in Table 5.11.
5.12. Rotation dependency

**Figure 5.8.** Initialization with undersampling and without rotation invariance. On the top row pixel $p$ maps to the green pen. When introducing a good candidate pixel by adding another piece of paper, the resulting initialization improves. Left: The original image, middle: inpainting after initialization only, right: mapping after initialization. The parameters used are in Table 5.12.

**Figure 5.9.** The shadow remains after removing the physical object from the image. The parameters used are in Table 5.13.
5. Results

*(a)* Original tea cup image. *(b)* Masked tea cup image.

*(c)* Original orientation *(d)* Horizontally mirrored *(e)* Vertically mirrored

*(f)* $90^\circ$ rotation

**Figure 5.10:** The results depend on the image orientation. Using the tea cup image taken in the AR-mode. The parameters used are in Table 5.14.
Conclusion

In the introduction, we stated the goal of defining an algorithm that can provide real-time object removal on mobile devices. Our proposition is based on the PixMix algorithm by Herling and Broll. We propose using a gradient descent optimization that can be implemented on the GPU. In addition, we suggest a number of changes to make the algorithm more robust. For processing a video stream, we suggest adapting the mapping propagation introduced by Herling and Broll. We, however, could not verify the performance of this mapping propagation, because the time constraints on this thesis did not allow to implement it. Instead, we provide an implementation of the image inpainting pipeline including object tracking and an extension to video inpainting by treating every frame independently as image inpainting problem. The performance of the unoptimized GPU implementation is good enough to claim that using the proposed algorithm real-time object removal on mobile devices in the context of diminished reality is possible.

To the best of our knowledge, we are the first to propose an inpainting algorithm for the GPU along with an actual implementation that runs on Android, iOS and Windows devices. With the AR application, we also give an example of such an algorithm in the field of augmented and diminished reality.

Our algorithm can successfully remove objects, even entire buildings, from an image. The results look natural and adapt to the image texture and structure nicely. We can reconstruct areas as large as 10% of the image size without introducing artifacts.

Our modification to the Herling’s and Broll’s work introduced the problem of the mapping pointing into the source area. We overcame this by implementing and evaluating several strategies and choosing the best for the final results. The GPU implementation using Vuforia and Unity with its compute shaders was source of various impediments. Working with shader code is hard in general since debugging is tricky. New tools to simply this process have been published. Just by the time, we finished this project, a new version of RenderDoc was released, now including tools for debugging shader code line by line.
6. Conclusion

Working with a prototype implementation first turned out to be most helpful. Understanding and adapting the algorithm was significantly more simple using an implementation that can easily be changed.

A major limitation of our work caused either by a mistake in both implementations or a flaw in the algorithm discussed in Section 5.12. This causes a bias that is visible in some situations.

Our objective was to show that real-time video object removal of acceptable quality is feasible on mobile devices. We believe that the presented results support our claim that both quality and performance are good enough and can be further improved based on our propositions. We further think that this thesis covers all crucial aspects to achieving this and our GPU implementation serves as a useful template therefor.
Future Work

Based on our work, we suggest extensions and new approaches we consider promising.

As discussed in the conclusion, we did not implement the propagation of the mapping to the next frame as described in Section 3.2.4. We expect this step to further reduce the number of iterations required to get a good mapping since it could provide a good initialization of the mapping for all mipmap levels. The tracking information could also provide an estimate on how much a frame as changed, which helps to decide whether the mapping should be optimized on a low-resolution mipmap level or whether it is sufficient to refine on a high-resolution level.

To increase the frame rate, the mapping can be optimized on low resolution only for the first few frames and once a good and robust inpainting is found, the high-resolution refinement is made on subsequent frames. Though this results in frames of lower quality from time to time, the rate of displayed frames is less dependent on the progress of the optimization. This, however, requires a robust propagation of the mapping from frame to frame as discussed before.

When applying the algorithm to an augmented reality application, two issues remain unsolved. First, physical objects cast shadows. The shadow can currently not be tracked and thus not be removed. However, the geometry of the physical object is known and even its material can be modeled in a preprocessing stage. Based on this the direction of the incoming light can be estimated and subsequently also the shadow, given the object stand on a planar or otherwise known surface. This information can then also be used to render the shadow of the virtual object.

Initially, we hoped to use Vuforia to track the physical object itself, instead of a tracker image below the object. Unfortunately, Vuforia’s object tracking was not robust enough by far to provide a reliable object tracking. Nevertheless, object tracking would make an AR application more elegant and potentially more flexible because the object would not have to be in a specific position on the tracked base.

The PixMix paper suggests an additional color cost term shown in Equation (2.4) that keeps the
mapping function coherent with a reference frame. This could stabilize the sensitivity to the noise in the camera frames.

A modification we consider very promising is the introduction of full rotation and reflection invariance. The spatial cost term has already been adapted accordingly which was a simplification to how the gradient is computed. For the color cost term, however, all rotations and reflections need to be computed. We consider computing the horizontal and vertical mirroring, as well as the $90^\circ$ rotations reasonable. This will increases the number of potentially good matches significantly.

To further reduce the risk of getting stuck in a local minimum, several candidates can be stored for the mapping. The initialization outputs the $n$ best candidates for each pixel and the optimization is performed on all of them. After some time bad candidates can be dropped and the most robust mapping chosen. A very similar approach has been suggested for PatchMatch [BSGF10] on which the optimization used in PixMix is based.

Finally, the results using the lausanne2 image often show a small difference in color intensity for the target and source area. This can be corrected, for example using seamless cloning as suggested by Pérez et al. in their work on Poisson image editing [PGB03].
Sample Images

This chapter contains all images used in the result section along with their masks. All images are named and this name is used as reference in the entire thesis. If multiple masks were used for the same image, we enumerate them, starting with zero (e.g. mask 0).

A.1. Own Images

These images were taken by Niclas Scheuing for this thesis.

A.1.1. building

![Images of building (a) building image, (b) Mask 0, (c) Mask 1, (d) mask2]

*Figure A.1.: The image we call building and its three masks.*
A. Sample Images

A.1.2. whiteboard

![Building image](image1.png) ![Mask](image2.png)

**Figure A.2.** The whiteboard image and its mask.

A.1.3. bunny

![Bunny image](image3.png) ![Mask](image4.png)

**Figure A.3.** The bunny image and its mask.
A.1. Own Images

A.1.4. *mill*

Figure A.4.: The mill image and its mask.

A.1.5. *zucchini*

Figure A.5.: The zucchini image and its mask.
A. Sample Images

A.1.6. tea cup

![a] tea cup image
![b] Mask

*Figure A.6.:* The tea cup image and its mask.

A.1.7. black and white

![a] building image
![b] Mask

*Figure A.7.:* The whiteboard image and its mask.

A.2. Images by Others

The images in the following sections have not been taken by the authors. Their creator is cited in the respective sections.
A.2. Images by Others

A.2.1. lausanne1

This image is by [LdpdheL].

![Figure A.8: The lausanne1 image and its mask.](image)

A.2.2. lausanne2

This image is by [lm].

![Figure A.9: The lausanne2 image and its mask.](image)
App Documentation - Render Modes

The implemented augmented reality application provides several debugging visualizations labeled Rendering Modes. This chapter serves as a documentation to what these modes display and how the information is encoded.

The top left drop-down provides several kinds of visualization.

**Solid** : Shows the camera image and the animated bunny rendered. The area to be inpainted is overlaid in blue.

**Mapped** : This is the usual way of using the App. The hole is filled with the computed colors and the bunny can be rendered by enabling the Virtual Object toggle.

**Error** This mode shows the magnitude of the cost function and its terms encoded into the color channels.

- red Total cost
- green Color cost term
- blue Spatial cost term
- alpha Constant 1

If the pixel is not mapped, it’s shown as gray. The color and spatial errors are multiplied with the corresponding $\alpha$ weight.

**Mapping** This visualizes the mapping texture.
B. App Documentation - Render Modes

red  x component
green y component
blue Constant 0
alpha Constant 1

**Background**  In this mode, the camera stream without any modifications is rendered.

**Update**  This is a visualization of the magnitude of the cost gradient and the cost term gradients.

red  Color cost term gradient magnitude
green  Spatial cost term gradient magnitude
blue  Constant 0
alpha  Total cost gradient length

The color and spatial components are multiplied with the $\alpha$ weight and normalized.
Bibliography


[BZ17] Connelly Barnes and Fang-Lue Zhang. A survey of the state-of-the-art in patch-
References


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Real-time hiding of physical objects in augmented reality

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
Scheuing

First name(s):
Niclas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Ltigen, 03.08.2018

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.
Project Description Master Thesis

Real-time hiding of physical objects in AR

Niclas Scheuing

Introduction
The goal of the project is to develop a method to remove a physical object in augmented reality on a mobile device through real-time in-painting. The object shall be a small-scale sculpture, and the method will replace this object with an estimation of what lies behind it.

Task Description
The tracking will be provided by Vuforia (3D tracking or marker-based tracking) and the method demonstrated on a test app built with Unity, in which the sculpture will be replaced by an animated CG version of itself. The method itself will at least partially run on shaders on the mobile GPU.

The 3D model will be given and already animated. The project will explore the problem in three phases: for static color background, any static background, and dynamic background. The expectations are: on high-end mobile devices, non-perceptible in-painting on static color background for a limited viewing arc with high frame rate, and visually continuous in-painting for static and dynamic background with acceptable frame rate.

Milestones

<table>
<thead>
<tr>
<th>Task</th>
<th>Due date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>12-Feb-2018</td>
</tr>
<tr>
<td>Read related work and implement app framework including tracking, animated character and dummy custom shader</td>
<td>12-March-2018</td>
</tr>
<tr>
<td>Design benchmark and collect dataset</td>
<td>9-Apr-2018</td>
</tr>
<tr>
<td>In-painting of simple static color background</td>
<td>7-May-2018</td>
</tr>
<tr>
<td>Midterm presentation</td>
<td>14-May-2018</td>
</tr>
<tr>
<td>Preliminary report structure</td>
<td>28-May-2018</td>
</tr>
<tr>
<td>In-painter of static background</td>
<td>4-June-2018</td>
</tr>
<tr>
<td>In-painter of dynamic background</td>
<td>2-July-2018</td>
</tr>
<tr>
<td>Preliminary report</td>
<td>30-July-2018</td>
</tr>
<tr>
<td>Final report</td>
<td>12-August-2018</td>
</tr>
</tbody>
</table>

Remarks
A written report and an oral presentation conclude the thesis. The thesis will be overseen by Prof. Robert W. Sumner and supervised by Dr. Stéphane Magnenat and Dr. Fabio Zünd.