Real-Time Virtual Viewpoint Generation on the GPU for Scene Navigation

by

Shanat Kolhatkar

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.Sc. degree in
Electrical and Computer Engineering

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Abstract

In this thesis we present a method for achieving real-time view interpolation in a virtual navigation application that uses a collection of pre-captured panoramic views as a representation of the environment. In this context, viewpoint interpolation is essential to achieve smooth and realistic viewpoint transition while the user is moving from one panorama to another. In this proposed approach, view interpolation is achieved by first computing the optical flow field between a pair of adjacent panoramas. This flow field can then be used by the view morphing algorithm to generate, on-the-fly, virtual viewpoints in-between existing views. Realistic interpolation is obtained by taking into account both scene geometry and color information. To achieve real-time viewpoint interpolation, a GPU implementation of the viewpoint interpolation algorithm has been developed. We ran our algorithm on multiple interior and exterior scenes and we were able to produce smooth and realistic viewpoint transitions by generating virtual views at a rate of more than 300 panoramas per second.

Acknowledgements

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work was developped within the scope of the NAVIRE project (virtual NAVigation in Image-based representations of Real world Environment) that aims at achieving real-time navigation in image-based renditions of real environments. The project covers all aspects of the image-based environment representation and navigation problem:

- the acquisition and storage of high quality panoramas
- the estimation of the pose between panoramas
- the interpolation of views
- the rendering and navigation tools

Our goal is to provide the user with an intuitive software to navigate any given environment in real-time, and provide the highest degree of realism and immersion possible. This work presented in this thesis constitutes an important step towards this goal by providing a real-time navigation application, but has the current drawback of having slight artefacts during the transitions.

There has been a lot of interest from the computer graphics and the computer vision fields towards image-based modeling and rendering methods [19]. Virtual navigation is one important application of such research, including virtual tours of tourist places, museums, as well as real-estate virtual visits. Any kind of application where achieving photorealism is a goal can actually benefit from advances in this field.

Recent virtual navigation applications use collections of images in order to allow users to remotely explore a given existing environment. The Google Street View system is a good example of a large-scale image-based model of city landscape. The system by

Furukawa et al. [56] is another example of virtual navigation system; this one focusing on virtual tours of indoor spaces. In order to improve the quality of the virtual exploration experience in an image-based model, several aspects could be improved. Exploration can be made more immersive by improving the overall quality of the images shown to the user. Using very large display (e.g. CAVE) or head-mounted display is a solution of choice when these are available. The recourse to panoramic imagery rather than regular limited field-of-view images also greatly contribute to the quality of the experience. Virtual exploration also implies the possibility of moving inside the environment. A multitude of points of view must then be available; too often these are limited to a linear path that the user is forced to follow during his exploration of the site. In order to visit the scene, users have to hop from one panorama to another through a series of mouse clicks. Such saccadic motion is unavoidable when the panoramas are far from each other and this greatly reduces the quality of the immersion and can make the user loose his sense of direction in the environment. To alleviate both problems, the user should be able to see the motion that took place to transition between viewpoints. This is a difficult problem since the relative position of the panoramas might not be known and no depth or geometrical information about the scene is generally available. Because of this, it is difficult to create realistic transitions between pairs of views, especially since the movement of the scene objects depends on their relative distance to the point of view.

Several methods have been developed to solve the problem of smooth and realistic viewpoint transition. These are based on video transitions, interpolated views and/or 3D modeling. In the simplest case, videos from one viewpoint to another are taken during the capture of the scene. This allows for transitions of good quality, but brings many other problems, such as the space required to store such videos, the limitation imposed on the ways a user can transit from one location to another and finally the necessity to re-acquire every possible transitions from a given view to a newly added view. Interpolation between pairs of views can be done using different schemes. Using simple linear interpolation of pixel intensities to generate view transitions is a straightforward solution but to produce realistic transitions, more sophisticated approaches are required. Finally, 3D modeling approach aims at fully reconstructing the environment, either manually or automatically, such that viewpoint interpolation becomes a 3D scene re-projection task. Creating dense 3D models of natural scenes is however very costly; consequently this is not a practical approach.

In this thesis we focus on the second type of approach to solving the transition problem, i.e. using view morphing to obtain realistic virtual views without explicit 3D

reconstruction. In addition, an essential requirement of our virtual navigation application was to come up with a fast view interpolation process such that views can be interpolated at frame rate. Users will then be able to realistically transit from one panorama to any adjacent one. 360° panoramas are used in this application in order to give to the user a higher quality immersive experience.

The scene is assumed to be static (i.e. does not contain moving objects) and the pre-captured panoramas are taken at a relatively short distance from each other. No 3D information about the scene or the motion is available.

1.1 Thesis Objective

This thesis addresses the problem of generating smooth transition between panoramas in the context of image-based virtual navigation in a remote environment. The ultimate goal is to allow the user to navigate in the environment freely and smoothly, without him being able to notice a difference between captured frames and generated frames. Our methods improves on the currently available methods by providing real-time navigation along a path, and provides a design that will allow the quality of the interpolation to follow the enhancements provided by advances in the research in the field of optical flow calculation.

1.2 Thesis Contribution

Our thesis brings the following contributions to the fields of optical flow, viewpoint generation and scene navigation:

- Most importantly the calculation of new viewpoints in real-time using a view morphing algorithm
- The ability to navigate an environment in real-time through the use of GPU programming
- Enhancing the optical flow to reduce the number of noticeable artifacts by applying few heuristic corrections

Our thesis work has spawned a couple of papers published in renowned conferences:

- "Real-Time Virtual Viewpoint Generation on the GPU for Scene Navigation", CRV 2010 [22]

- "Interactive Scene Navigation Using the GPU", CGI 2010 [21]

1.3 Thesis Outline

In Chapter 2 we presents the previous works relevant to our method. We will first present the most important works in image interpolation and virtual navigation. Second there has been a extensive work done in the field of optical flow calculation between two planar images, and we will present the important works that has been done in this domain. Finally we will present the previous work that has been done regarding the epipolar geometry estimation between pairs of planar images and between panoramas.

In Chapter 3 we explain how the optical flow calculation algorithm that we have selected to use for our software works. We also describe in greater details its matching function and its goodness function, which allows it to retain the important features for which it has been selected: contrast-invariance matching and slanted surface handling.

In Chapter 4 we describe the extension of the optical flow algorithm to cubic panoramas. First we will describe our extended cube representation for panoramas, which allow us to solve most boundary problems that were present in the usual cubic panoramas. Next we describe our smoothing correction step, which aims at removing the irregularities within a dense optical flow field. The next section is aimed at describing our improvements to the dense optical flow, which uses the epipolar geometry between panoramas.

In Chapter 5 we describe the view interpolation algorithm. We will first describe the algorithm on which our method was based. This algorithm was created to allow the user to mix multiple textures of different types to create new ones. We will describe the important features of this algorithm, namely the warp computation which is used to calculate the dense displacement field between two given textures, and the morphable interpolation which create the interpolated images by taking into account both the color information and the displacement information. Finally we will describe our own view interpolation algorithm which was developped for pairs of real panoramic images.

In Chapter 6 we provide the implementation details required to achieve real-time navigation and calculation of the transitions. There are four important implementation requirements to accomplishing the navigation in real-time which are the precalculation of the optical flow, which can take up to an hour to compute for each pair of panoramas, the buffering of the data that will be accessed in the next few moves, the multi-threading of the application which allows us to access the hard-drive and load the necessary textures at the same time as the interpolation, graphics and input handling calls are made; and

finally the implementation on the GPU of the morphing algorithm, which allows us to use the power of the graphics card, which outshines even the most powerfull multi-processor CPUs.

Our study ends with the presentation of some of our results (Chapter 7) and by concluding on the ramification of our method and some possible improvements to make our navigation software more immersive and the increase the quality of our results (Chapter 8).

Chapter 2

Related Work

The solution we propose for view interpolation has its roots in the view morphing and optical flow research fields. This chapter reviews the previously published works in these two areas. Our thesis work also bring forth improvements to the currently available applications in the field of virtual navigation, and we will also discuss the relevant work of this field as well. Finally we will introduce a couple of optical flow field correction steps based on the epipolar geometry between a pair of images, and a summary of the previous advances in this field will also be necessary.

2.1 View Morphing and View Interpolation

There are a few different approaches to texture morphing that have been developed over the last few years amongst which are [58] and [32]. The first paper depends heavily on user input, and only works for textures that are composed of repeated similar patterns (for example cells of the interior of a bee hive). The second paper represents, to the authors' knowledge, the state of the art in the field of texture morphing, and combines linear color interpolation and motion compensation to generate the composite texture. This paper is at the basis of our interpolation algorithm. It has the advantages of not depending much on user input and of working with a wide variety of textures while still producing high quality results. However, this method needs to have features of similar sizes in both origin and target textures. Also, this method cannot morph smoothly between highly different textures. These disadvantages are not relevant to our approach since we use real life images, which are highly structured, and our images are highly correlated since they are taken along the path followed by our camera.

2.1.1 Virtual View Generation

Virtual View Generation can be considered as a type of texture morphing, where we use real-life images to generate new viewpoints. We will describe some important previous works in this section.

One of the base paper in the field of view interpolation is [42]. This paper focuses on creating the transition views to move between pairs of images, which are not necessarily parallel. To create the transition images, the method prewarps two input images to fit them on parallel planes. It then applies a morphing procedure to obtain an intermediate image and postwarps that intermediate image to obtain the resulting transition image. Both prewarping and postwarping transformations are done using projective transformations. A downpoint of this approach is that it cannot handle complex scenes. It was designed to handle pairs of images of single objects taken in different poses. This makes this approach difficult to generalize to arbitrary viewpoints. Other papers regarding view morphing also attack the subject of view interpolation, such as [23], [50]. Both approaches triangulate the images using a certain set of points in the images, which can be chosen manually or automatically using for example [46]. Once the triangulation is done, the triangles of the images are warped and the colors are interpolated to generate a new virtual image.

A more recent work related to virtual navigation, was done, like this thesis work, as part of the NAVIRE project [8]. Their approach uses ray-tracing to synthesize new views from an input pair of cubic panoramas. One of the input panoramas is selected as the reference panorama and the the position of the other input panorama is evaluated relatively to it using methods presented in the Section 2.4. As well the position of the virtual viewpoint is defined relatively the reference cubic panorama. Finally, for each pixel in the virtual view panorama texture, a ray is cast and its intersection with both input panoramas is calculated. The resulting color for the pixel is the interpolated value of the two intersected pixel. This approach is time consuming because of the high number of rays that need to be cast for each virtual view texture. The resulting interpolated images also presents many artifacts because, as we explain in later, the evaluation of the relative position of the panoramas using the color information only is error prone, and because the colors of the resulting images are interpolated multiple times. (once within the image since it is highly unlikely that the ray will intersect with exactly one pixel, and then between the intersected pixels within the two images: thus 8 interpolations maximum)

The work presented in [45] was also done as part of the NAVIRE project. Their work in virtual navigation focussed on the creation of very short translation movements in cubic environment maps. The transition cubes between two viewpoints are created by warping the different faces of one of our input cubic environment map depending on the movement direction. For the first half of the transition between image I_1 and I_2, they warp the faces of cube I_1 onto the transition cube, and for the second half of the transition, they warp using only cube I_2. To calculate the warping coefficient, the method uses optical flow. It calculated the optical flow for every possible pixel displacement between 1 and the maximum allowed displacement between the images (80 pixels) and select as a global displacement the one setting the average norm of the optical flow as minimum. This method does not provide high quality results, or real-time navigation, and by only warping one of the two input cubes at a time, looses the information available in the other one.

In our approach, fluid view transitions are generated on the fly for the whole panorama which allows the user to continuously look around the scene while moving. The addition of new panoramas requires only the computation of the corresponding optical flow fields. The transition panoramas at intermediate viewpoints are then automatically generated.

2.2 Virtual Navigation

One of the most important paper of this field was developped by Apple Computer [7]. This paper laid the foundation of most navigation software by defining the necessary operation required for a user to navigate a virtual environment: rotating the camera, rotating around an object, moving the camera in the environment and finally camera zooming. Rotating the camera allows the user to look around the environment while staying at the same viewpoint. This is done by rotating the environment map which is used to display our panorama. Rotating around an object can be done in two ways, first using videos of the object or more interestingly through view interpolation techniques, which in their case is done using [44]. Looking at an object can also be done using the methods described in the previous section, principally [42]. Moving the camera in the environment can be done through rotating the camera, which was described earlier, and through translating it. The camera translation is done in one of two ways: through environment map movies, which are 360° movies that are taken in a certain path and allow the user to change the orientation of the scene. This approach requires a lot of disk space, and required the acquisition of every possible transition between panoramas and

thus is not very flexible. The second way to achieve camera translation is through the interpolation of panoramic images. In this case, the method is to interpolate new views from the nearby panoramas in order to generate a smooth path. Finally, the camera zooming feature is generally achieved by having multiple environment maps of different resolution for the same scene. When the user zooms on a certain part of the scene, the program interactively selects the environment map resolution that gives the best quality, and provide interactive speeds while zooming. This can also be used to give the user a feel of moving forward or backwards inside the environment. In Apple Computer's paper, the camera translation was done through a series of jumps from one cylindrical environment map to the other.

Another interesting previous work is presented in [48]. This work is more of an image indexing method, since the method aims at grouping together images of the same place and at allowing the user to navigate through them. To do so, it identifies points of interests and assembles images that are similar. It then estimates the relative viewpoint of these images by generating a 3D point cloud from the point of interests in the image. Then it places these different viewpoints in a virtual map to allow for navigation and to track the user position easily. Finally, the user is allowed to navigate through the images by selecting images on a virtual map, by selecting a region of interest in the currently seen image or by selecting images with the same viewpoints. Once the navigation is initialized, the method transitions between the images by using a simple plane based morphing algorithm. This method works with standard planar images, and the transitions between the different photographs present some artifacts but the image indexing is impressive.

A good example of 3D reconstruction can be seen in [41]. Their approach aims at reconstructing the scene in 3D using a high number of images. The approach however works only on simple scenes. A good advantage of this approach is that it allows the user to freely navigate the scene by allowing 3D movement instead of being limited to a unique path as in the other approaches. A more recent work in reconstructing the environment is presented in [56], and is aimed at reconstructing indoor environments. The method presented in this paper reconstruct the environment automatically from an input set of images of the environment by using structure from motion [47] and multi-view stereo [9] to calibrate the cameras and retrieve a set of 3D points which represent an initial reconstruction of the environment. Once this is raw reconstruction is available, a depth map is generated using the "Manhattan-world"' assumption [55], which states that the world is highly structured, with the planes aligned with the X, Y and Z axis. Finally the depth maps and the multi-view stereo points are combined

together to generate the final 3D model of the scene, at which point 3D navigation is possible. One of the biggest challenges with this work is that most of the environment is composed of planar, textureless scenes. However, it gives very impressive results and the reconstruction of the environment is completely automatic and seems quite accurate. Some of the downpoints of this approach are that this method only supports axis aligned surfaces, fairly small environments and presents some artefacts during the navigation.

In recent years, publicly available application have been developped to achieve virtual navigation over the internet. Both of the approaches that we mentioned previously are represented by such software. Google Street View and Everyscape, use panoramas to represent the environment and linear interpolation between pairs of images as transitions. The Google Street View method is the most similar to our own system: a panoramic camera such as our own Ladybug is mounted on a series of vehicules (car, bike, van) and driven around the city taking panoramic pictures as set intervals. The images are then threated to remove private elements such as faces or vehicules licence plates and then uploaded to a server. To create the transitions, Google Street View applies zooming on the face of the portion of the panoramas currently being viewed to simulate movement as well as linear interpolation to transition between the images.

Bing, VillesEn3D and Google Earth model the different buildings in 3D and place them on a world map and allow the user to see the environment from the sky. VillesEn3D is of higher visual quality than Bing and Google Earth but is only available for thirteen cities in France and three in Spain whereas Bing and Google Earth model cities from around the world. In all these applications the buildings are modeled using 3D modeling programs (such as 3DS Max, Maya or proprietary tools) and placed on a world map. In the case of Bing and Google Earth, the buildings are not textured whereas in the case of VillesEn3D textures are present on all buildings and were most probably created by artists. Creating the buildings in 3D is time consuming, and using artists to create the textures for each building increases even more the time necessary to accomplish such a software. These sotwares either use artists to model the building or ask for the help of the users, but no automatic method is currently available, unlike the Google Street View type of applications.

2.3 Optical Flow

The optical flow is calculated on pairs of images taken from different points of views of the same scene. The optical flow is calculated per pixel and represent the movement

that this pixel takes when moving from the first image into the second image of the pair. In most images, some occlusions occur, and in such cases the optical flow for that pixel cannot be calculated accurately. The optical flow can be calculated for selected pixels that are more easily identified in a scene, or for every pixel in the scene, in which case it is called the dense optical flow of the pair of images.

Many approaches to calculate the optical flow between pairs of planar images have been proposed. One of the biggest problem when creating an optical flow algorithm is that there is no easy way to compare the optical flow produced, because the ground truth between pairs of real images is difficult to produce. Scharstein and Szeliski [40] studied some of the most important algorithms developped before 2002 and introduced a way to compare different optical flow algorithms. Their website http://vision.middlebury.edu/stereo/ makes the evaluation routines and datasets freely available, and allows the user to post their results and compare them to other methods. These results are freely available to anyone and allows a uniform method of comparing optical flow results. The number of datasets however is restricted, due to the facts that we mentionned earlier.

Amongst the wide variety of algorithm available, some of the best performing ones are [3, 53, 54, 37]. A few very interesting ones because of the fact that they calculate the optical flow in real-time, but which perform less good according to the middlebury test framework are [10, 43, 18]. Some of the more classic methods are [14], [28], [6]. These methods are based on matching the pixel neighborhoods of the origin image with different candidate neighborhoods of the target image in order to determine the best possible match. Another more recent approach to calculating the optical flow is [36]. This is the algorithm we will use in this thesis work and it is presented in details in Chapter 3. It focuses on calculating a goodness measure for each of the pixels, which relies on how similar the displaced origin pixels are to the ones of the target image. This algorithm requires the user to define an interval, and all possible displacements will be tested to find the best solution. No optical flow methods gives perfect results, partly due to the fact that there are no infallible way to compare neighborhoods in a truly viewpoint invariant way. The neighborhood of a same pixel changes depending on how far it is from the view point, and this makes predicting the deformation of the neighborhood very difficult.

2.4 Epipolar Geometry Estimation

The epipolar geometry of a pair of images represents the projective relation between both images of the pair. It allows us to know the position of the two images with respect to each other. The epipolar geometry can only be estimated because the algorithms currently available are not entirely accurate due to possible mistakes when matching images together. As well, the actual length of the optical flow vectors depends on the distance from our viewpoint to the pixel of the panorama, and thus it is not possible to get such information when only the color information of the scene is available. Thus the epipolar geometry information is, given the current methods, a useful tool to constrain the displacement vectors in rigid motion when allowing a certain margin of error.

Estimating the epipolar geometry between pairs of panoramas has also been a area of research that has received a lot of attention over the years. A lot of different algorithm are available to the user such as the ones presented in [11], [24] and [13, 34]. The latter is considered by many to be the state of the art in the field. A breath of other methods are available, and reviews of these technics can be found in [52, 15]. However most of these methods are specifically designed for limited field of view planar images. Other methods present solution to the calculation of the optical flow on omnidirectional camera such as our Ladybug by representing the environment as a sphere [51, 25]. The paper [26] also tackles the problem of estimating the epipolar geometry for omnidirectional cameras by using as input a calculated optical flow at antipodal points of the image sphere. To achieve this, the authors present two algorithms, one using the RANSAC framework [31, 38] and the other performs the Hough-reminiscent voting [16]. This is the inverse process that we used in our epipolar constraining of the optical flow algorithm, where we use the epipolar geometry of the scene to try to enhance the result of our optical flow.

On top of estimating epipolar geometry of the scene through the previously mentioned methods, an epipolar rectification scheme can be applied to the results to enhance them. Such schemes have been developped for 2 views using uncalibrated cameras [12, 27] or on 3 views with calibrated cameras [30] or uncalibrated ones [49, 57]

Work has been done previously in the VIVA lab to estimate the epipolar geometry between 2 panoramas [20] and we will be using the software developped in their thesis to estimate the epipolar geometry of our scene. This approach works on 3 view cameras and is, as said by the authors of the method, based on the method [17] and applies a distortion reduction scheme. This technic also uses the previously mentioned RANSAC framework.

2.5 Conclusion

Our thesis work touches different fields of computer vision and computer graphics. In this chapter we have presented the related works that are relevant to our work. In the next chapters we will go in more details on the view morphing and optical flow calculation algorithms that our method was based on or uses, as well as the different improvements that we brought to the different methods to increase the results in the case of real life panoramas.

Chapter 3

Optical Flow Estimation

As we mentioned before, our virtual view point generation system uses the optical flow field between reference images as an input to the interpolation process. In this Chapter we will describe the optical flow algorithm that we use in our approach.

3.1 Basic Algorithm Description

Our view interpolation approach is based on the computation of the optical flow field between the reference images. We use the method presented in [36] and for which an implementation is publicly available[1]. The user needs to input an interval for the possible values of the displacement vector in x and y axis, and all possible combinations of these values will be tested by the algorithm in order to find the best displacement vector for each pixel. For each of these shifts, the algorithm computes a goodness function for each pixel of the image. The goodness function is calculated as the sum of two one-dimensional functions: one that is calculated on the line parallel to the displacement vector, and the other on the line orthogonal to it. The goodness value for a pixel $p = (x, y)$ can be interpreted as the number of pixels connected to $p' = (x + dx, y + dy)$ that have an intensity similar to the ones in the target image, with (dx, dy) being the candidate displacement. Once the goodness function has been calculated, the best match is selected to decide which displacement applies best to p.

We selected this algorithm because it generally gives good results. Some of these results are shown in Table 3.1 and were retrieved at the time of the writting of the paper [35]. We only show the highest, second highest and lowest results in this table

[1] http://www.cs.umd.edu/ ogale/download/code.html

Rank	Algorithm	Tsukuba			Sawtooth			Venus			Map	
1	Layered	1.58	1.06	8.82	0.34	0.00	3.35	1.52	2.96	2.62	0.37	5.24
2	Belief prop	1.15	0.42	6.31	0.98	0.30	4.83	1.00	0.76	9.13	0.84	5.27
6	Ogale	1.77	0.95	9.48	0.61	0.17	5.05	3.00	5.22	7.63	0.21	3.01
28	Max surf	11.10	10.70	41.99	5.51	5.56	27.39	4.36	4.78	41.13	4.17	27.88

Table 3.1: This tables shows a comparison of the optical flow algorithm that we used (Ogale) with other already existing optical flows. These results were retrieved using the middlebury test set provided in [35]. All the algorithms were ran on a set of four pairs of images. Each pair of images had its associated ground truth displacement which was manually defined. The result values in each cell of the table was obtained by comparing this ground truth to the result obtained by running each optical flow algorithm on every pair of images.

as a measure of comparison. The algorithm ranked 6 out of 28, with error percentages ranked from 2 to 20. But the most interesting aspect of this algorithm are the features that it provides and that we will describe now: it handles well occlusions, and it uses a matching measure that is contrast-invariant. This is an important feature for outdoors scenes where the lighting conditions of the scene changes between the different views. Another point that influenced our decision is the fact that this algorithm can identify slanted surfaces and consistently compute the corresponding optical flow. This is a particularly interesting feature in the case of urban scenes where planar building facades are frequently observed. The last interesting point of this method is that it is able to identify slanted surfaces.

3.2 Theory discussion

In this section we will describe the theoretical aspects of this algorithms: the general idea of the paper and the slanted surface identification.

The general idea of the paper is straightforward: it is given a series of possible shifts for the image, and it tries to identify all the connected components between pairs of images. Each pair of image is created by using the origin image shifted by a certain vector and the unshifted destination image. There are usually multiple connected components in a pair of images, and they are the biggest number of pixels that are similar in both images given a certain matching function. An important point to the identification of connected

components is that if it is traversed by a horizontal edge, it is considered to be in fact two different connected components. The matching function is defined by the user, and we will describe the one used in the algorithm in the next section.

3.2.1 Slanted Surface Handling

This method was developped for stereo vision primarily and it recognizes two types of slanted surfaces (or slants): horizontal and vertical. This methods makes a distinction between vertical and horizontal slants because vertical slants can also cause horizontal disparity during the comparison but the horizontal slants can't because of the fact that it is considering stereo vision. The important observation to be made about slanted surfaces, is that a same slant always has different size in different images if the position of the viewpoint changes. This means that the uniqueness constraint which many optical flow obey (a pixel in an image corresponds to exactly one pixel in the other image) does not hold and need to be modified accordingly. In this section, we will first describe how horizontal slants are identified and then we will do the same for vertical slants.

Horizontal Slants Identification

To identify hozitontal slants, both images are stretched in turn to cover the space of all possible slants (an interval of horizontal pixels). To establish a correspondence between the two images all the stretched versions of each images are used, and the line segments are compared to each other, instead of pixels. When the line segments cannot be match in this step, the method assumes that it detected an occlusion.

The way to match the pixel intensity when considering slanted surfaces has been changed from the usual algorithm pixel matching algorithm presented in [5]. Each scanline is resampled by stretching one of them to match the horizontal slant that we are considering. Once it is done, this stretched scanline is matched against the another unstretched scanline in the other image using the usual method. This comparison method works as follows (using the notation in the author's paper): Given two pixels x_L and x_R to be compared to each other, in the scanlines $I_L(x)$ and $I_R(x)$ respectively. In the case of slanted surfaces, pixel x_R is defined as: $x_R = mx_L + d.$ and the following are set:

$$I_L^{min} = minI_L(x_L - \frac{1}{2}), I_L(x_L), I_L(x_L + \frac{1}{2}) \tag{3.1}$$

$$I_L^{max} = maxI_L(x_L - \frac{1}{2}), I_L(x_L), I_L(x_L + \frac{1}{2}) \tag{3.2}$$

$$I_R^{min} = minI_R(x_R - \frac{1}{2}), I_R(x_R), I_R(x_R + \frac{1}{2}) \tag{3.3}$$

$$I_R^{max} = maxI_R(x_R - \frac{1}{2}), I_R(x_R), I_R(x_R + \frac{1}{2}) \tag{3.4}$$

and subsequently:

$$d_L = max0, I_L(x_L) - I_R^{max}, I_R^{min} - I_L(x_L) \tag{3.5}$$

$$d_R = max0, I_R(x_R) - I_L^{max}, I_L^{min} - I_R(x_R) \tag{3.6}$$

$$d = mind_L, d_R \tag{3.7}$$

The uniqueness constrain on a pixel basis is replaced by a uniqueness contraint between the different intervals of the the left and right scanlines, which means that each line interval LI_{left} in the left image can only correspond to one line interval in the right image LI_{right}. If a line interval LI_{left} is matched to multiple line intervals in the right image, the different interval matchings are corrected and the best match is selecting depending on the previous matching correspondence.

Vertical Slants Identification

There are multiple observations to use when differencing vertical slants from horizontal ones:

1. Vertical neighbors separated by a horizontal edge or no edge at all should not be connected.

2. Disparity can change even when there is no change in color or intensity, thus it cannot be assumed that disparity is vertically constant but that it is continuous.

3. Vertical neighbors lying on non-horizontal edges should be connected.

To identify vertical slants, the method first detects intensity edges using a standard Canny edge detector as well as their directions by computing the gradient direction. It is also assumed that every pixel is connected by links to the pixels above and below him.

Using the previous statements, the only pixels that end up being connected are the pixels that lie on a non-horizontal line. Finally when labeling the connected components, if two vertical neighbors are linked, then they are assumed to be part of the same connected component.

3.3 Implementation discussion

In this section we will describe the algorithm specific part of the method such as the algorithm architecture, the contrast-invariant matching function as well as the goodness function used in this method.

3.3.1 Algorithm Architecture

The algorithm works as follows: The user inputs two images I_1 and I_2, as well as the search intervals along both x and y axis: [minX, maxX] and [minY, maxY].

The algorithm exhaustively checks every possible combination of the values defined by the user for x and y (pair of shift vectors). For each of these pairs, it does the following:

for $Shift_x = minX$ **to** $maxX$ **do**
  for $Shift_y = minY$ **to** $maxY$ **do**
    - Compute the matching between the two input images using $Shift_{xy}$ to shift the target image. 3.8
    - Compute the Horizontal Goodness (HGoodness) function for the whole image 3.12
    - Compute the Vertical Goodness (VGoodness) function for the whole image 3.12
    - The total goodness of image I_1 is the product of the horizontal and vertical goodness that we just calculated:
    $TotalGoodness = HGoodness * VGoodness$
    - For each pixel in the image, if the goodness calculated for the current shift is better than any other that was calculated this far for both left and right images, we use its corresponding shift vector as the best possible displacement for this pixel.
  end
end

3.3.2 The Matching Function

The matching function $M_{x,y}$ between 2 images is calculated as follows:

$$M_{x,y} = e^{-(\alpha/255)} * mean(|Match(Shift_{x,y})|) \tag{3.8}$$

$Shift_{x,y}$ is the candidate shift vector, and mean is a function that works as follows: for every pixel in the image $p_{i,j}$, its value is the average of all the values of its neighbors in a certain neighborhood size:

$$p_{k,j} = \frac{1}{n * m} \times \sum_{i=1}^{n} \sum_{j=1}^{m} p_{i,j} \tag{3.9}$$

Where $p_{k,l}$ is the pixel at the center of the neighborhood of size (n, m) and $p_{i,j}$ represents the pixel of this neighborhood. In the algorithm that we used, α is set to 20.0, and n = m = 3.

For each pixel the following values are retrieved: m1, M1 which are the minimum (resp. maximum) of the values of the pixels in the neighborhood of the current pixel in image I_1. Identically, the values m2 and M2 are retrieved, and are taken using image I_2.

The Match function for a given pixel $p_{i,j}$ in image I_1 and its shifted equivalent $p'_{k,l}$ in image I_2 is calculated as follows:

$$Match = \frac{match1 + match2}{2} \tag{3.10}$$

$$match1 = \begin{cases} m2 - p_{i,j} & \text{if } m2 - p_{i,j} >= 0 \\ p_{i,j} - M2 & \text{if } p_{i,j} - M2 >= 0 \\ 0 & \text{in other cases} \end{cases} \tag{3.11}$$

We calculate $match2$ similarly, and we have $k = i + Shift_x$ and $l = j + Shift_y$.

This function is applied to the whole image, excluding a few border pixels depending on the shift that is considered.

3.3.3 The Goodness Functions

The goodness function relies on the matching function function. The goodness function $G(x, y)$ for pixel (x, y) is set as:

$$G(x, y) = (1 + G(r, s)) * M(x, y); \tag{3.12}$$

When calculating the horizontal goodness function, going from left to right, we have $r = x - 1$ and $s = y$. And when going from right to left, we have $r = x + 1$ and $s = y$. And as well, when calculating the vertical goodness function, top to bottom: $r = x$ and $s = y + 1$. And finally in the same way when going from bottom to top: $r = x$ and $s = y - 1$ As mentioned previously, two goodness functions are calculated for each pixel a horizontal goodness function and a vertical goodness functions. Each of these goodness function is the sum of two smaller function: a left and right goodness function for the horizontal one, and a top and down goodness function for the vertical function.

Once the left goodness function has been calculated for all pixels, we do the same for the right pixels, by looking at the image from right to left instead of left to right. And both left and right goodness for a certain pixel are summed together to give the total horizontal goodness value for this pixel.

The vertical goodness function is calculated in an identical way, except that we look at the pixel from top to bottom and then bottom to top.

3.4 Results

In Figure 3.1, an example of the flow field is displayed. The top image is the origin image and the bottom is the destination image, and the arrows represent the flow vectors for selected pixels. The arrows'size has been increased to be more readable. The flow vectors of the target panorama are the opposite values of the origin image's flow vector, this is due to the assumptions taken by this optical flow algorithm.

We assume in the following discussion that the image had n pixels. Calculating the Mean, m1, m2, M2 and M2 is calculated on each pixel and depends on a certain neighborhood size. Assuming that all the neighborhood sizes are the same and are square, and setting the size as s then each of the image calculation takes $O(ns^2)$ time. Given that all the information necessary to calculate the matching function has been retrieve through the previous step, the matching function is of constant time $O(c)$. To calculate the Horizontal Goodness function, the method goes through the image twice (left to right and right to left) and applies some matching function with constant time thus this step takes $O(2n)$. Similarly, calculating the Vertical Goodness function also takes $O(2n)$. The calculation of the Total Goodness for each pixel is the product of HGoodness and VGoodness and is calculated for each pixel, and thus is an $O(n)$ operation. To conclude, this algorithm takes $O((5 + s^2 + c) * n)$ to compute, where $(5 + s^2 + c)$ is a constant.

According to our results, to compute a 320x320x6 image with a [-40;40] interval for

both the x and y interval values of the shift, the calculation takes 1 hour. To compute on a 1280x1280x6 image with a neighborhod of [-160; 160] it can take up to 26 hours.

3.5 Conclusion

In this Chapter we have described the optical flow algorithm that we use in our image and we will be used as is, without any modification for our results. In the next Chapter we will introduce a few ways to improve the quality of the retrieved optical flow.

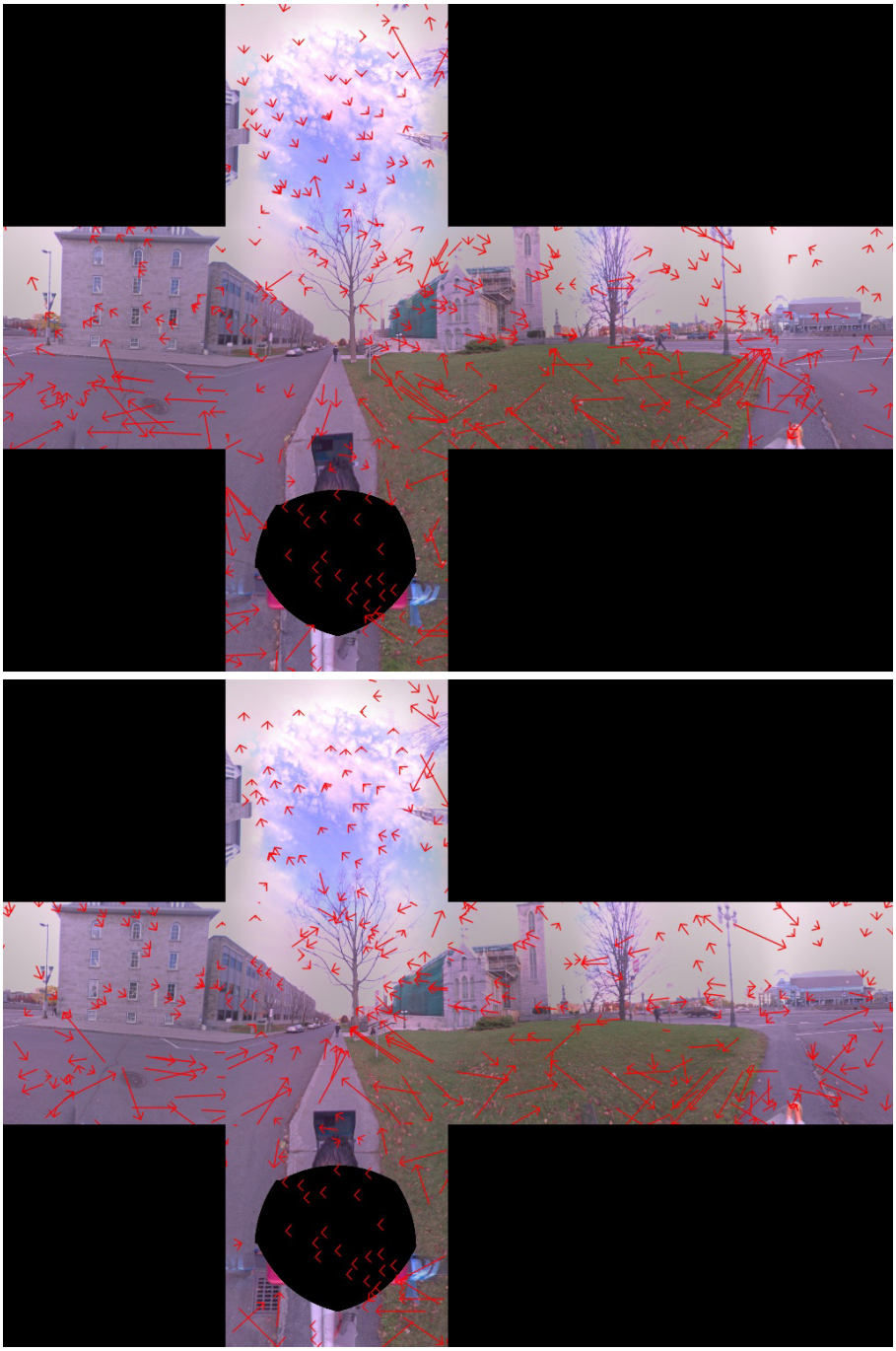

Figure 3.1: Example of an optical flow on a pair of panoramas. The top image is the origin image and the bottom is the destination image, and the arrows represent the flow vectors for selected pixels. The arrows'size has been increased to be more readable.

Chapter 4

Estimating the optical flow on panoramic images

Most current optical flow algorithms work with single planar textures, and our different correction steps were designed to work with such textures. However, to enhance the sense of immersion of the user and the realism of the scene rendering, we have chosen to use 360° panoramas. In this Chapter we will describe how to extend the currently available optical flow algorithms to work on cubic panoramas as well as a certain number of corrections steps that we experimented with to enhance the quality of the results.

Different representations are available to manipulate the panoramas: cubic, spherical or cylindrical panoramas (Figure 4.1). There are two downpoints to using spherical panoramas. The first one is that there is no way to represent a sphere with patches of same sizes. Depending on where you are on the sphere, you will have patches of different sizes, shapes and orientations, and this is especially obvious when you compare the patches of the equator with the patches of the poles. The second problem is that the spherical representation, partly due to what we previously mentionned, brings some noticeable deformations when looking around an image. When panning around, the image becomes distorted, and the object on the boundary of our view appear to be stretched. Such artifacts are very noticeable and reduce the immersion of a user and can even sicken some users. An example of this phenomenon can be seen in Figure 4.2, which is a picture of the Ta Prohm Temple in Angkor, Cambodia, and was taken from the website [2]. Both images in the picture were taken from the same position in the panorama, but the view direction is different in each of the image. When looking at the different elements of this scene, such as the man in white shirt, we can see huge

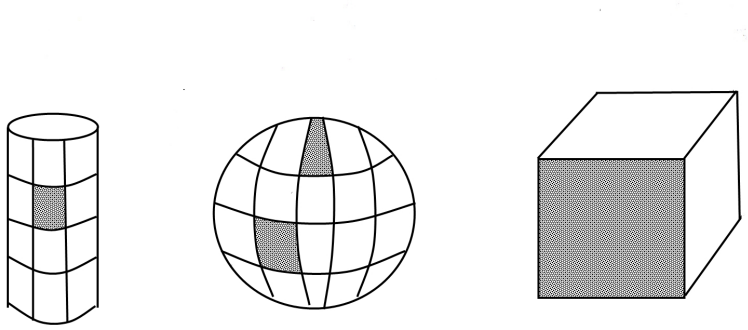

Figure 4.1: Representation of the different types of panorama representation: cylindrical (left), spherical (middle), cubic (right).

deformations of the object. One way to solve this issue, would be to zoom in the image, at which point the artefacts would be less noticeable, but this would severely reduce the field of view. This is not a valid solution as it would diminish the user's experience. Another possibility would be to increase the size of the different texture patches but would bring other kinds of problems such at the fact that we would be using an unconventinal geometric form for this type of application (such as an octahedron), and converting our scene to fit this shape would be fastidious.

The cubic and cylindrical panoramas, allow us to have patches of similar sizes on our front and side view, which reduces considerably the distortion effect that we see in the spherical panoramas.

Using cylindrical panoramas requires to create a special case for the top and bottom faces of the cylinder. Also when unfolded, cylindrical and spherical representations tend to introduce non-linear deformations that complicate the comparison of panoramas which is not the case with the planar geometry of the cubic representation: lines of a certain direction in a certain panorama can have a different direction in another panorama, even if the panoramas were captured very close to each other. This problem would require us to find an optical flow algorithm that would deal with such deformation. Using the cylindrical or spherical represensations would not allow us to use the breath of already available and very powerful optical flow algorithms

In cubic panoramas (see Figure 4.3), all 6 faces are identical; each face of the cubic panorama is a regular limited field of view image. Therefore, standard optical flow algorithm designed to work with such images will be directly applicable to each face of

the cube panoramas, and the only necessary extension to these algorithms is a way to deal with face transitions. As well, since the faces of the cube are identical and have the same size, there will be less optical deformation in the scene than with using another kind of panorama.

4.1 Extended cubic representation

Classical optical flow field estimation methods have been designed to work on regular planar limited field-of-view images. The main advantage of the cubic representation is that it allows to decompose the global 360° panorama into 6 regular limited field-of-view cameras. However the independent computation of the optical flow on the 6 cube faces would cause annoying artifacts at the boundary of each face. We have solved this problem by simply extending each face such that it becomes possible to correctly estimate the displacement of a visual point moving from one face to another. One such extended cube is shown in Figure 4.4 where it can be seen that some elements on one face are repeated on adjacent faces, and in Figure 4.5 we can see 2 example extended cubic panoramas displayed in 3D. Faces are extended such that image points close to the cube edges that would change face in a normal cubic representation, will end up inside the replicated area of the extended cube. This allows us to ensure that flow calculation can be done on each face independently, and still correctly estimate the motion of the pixels at each face boundaries. Thus, when calculating the optical flow on each face of the extended cube, the moving objects that are changing face in a normal cube and dissapear of our face, are actually still present in the extended cube's face.

Assuming that the maximum pixel displacement in the image is $(d_x,\ d_y)$, then the minimum extended cube face size should be $size_x + d_x,\ size_y + d_y$, where $(size_x,\ size_y)$ is the size of a normal cube's face.

Note that, when proceeding this way, it happens that some displacement vectors are computed twice, because they appear on each extended portion of two adjacent faces (see Figure 4.6). To ensure consistency of the results across the faces, we need to check which of these two solutions gives the best result by comparing the color neighborhood in both images according to the L2-norm. The one with the greatest distance is then replaced by the other one.

Figure 4.2: Example of the deformations happening when using a spherical representation for 360° panoramas. Even though the viewing position is the same and only the viewing direction varies, the size of the different objects is radically different. [2]

Figure 4.3: Example of a cubic panoramas, displayed in an unfolded form

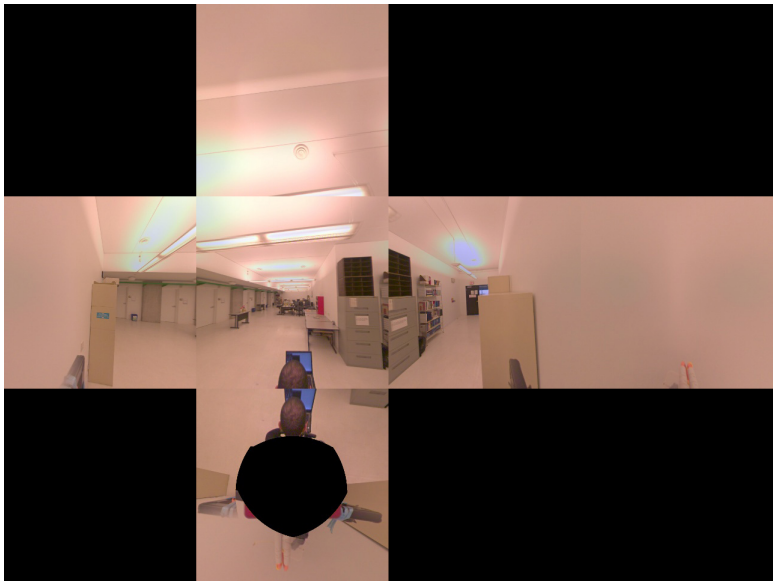

Figure 4.4: Example of our extended cubic panoramas, displayed in an unfolded form.

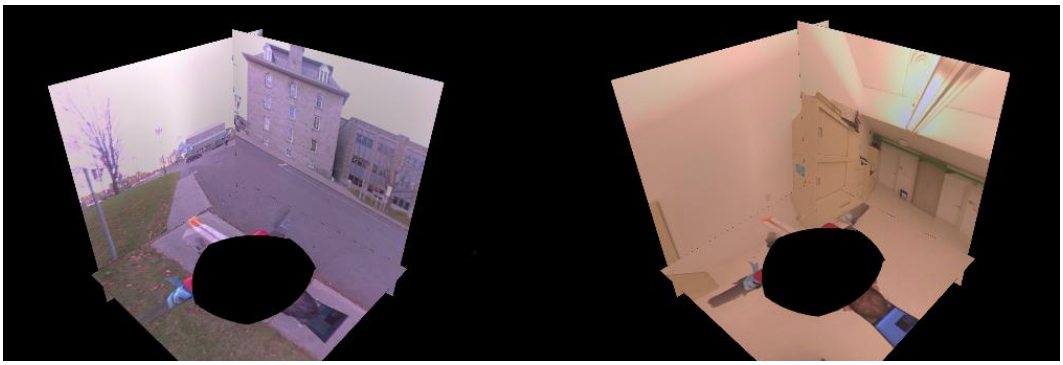

Figure 4.5: Example of our extended cubic panoramas, displayed in 3D, showing 3 faces.

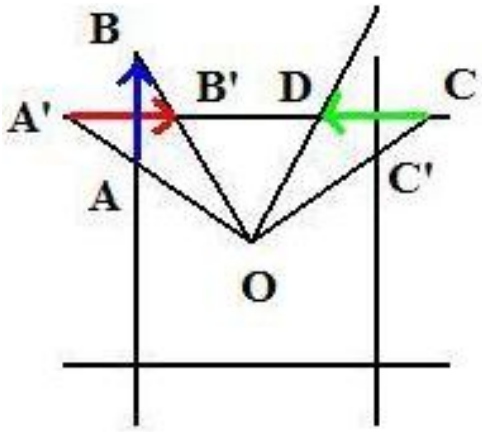

Figure 4.6: This figure illustrates the geometry of the extended cubic representation drawn in 2D. A' is the projection of A on a different face, and the vectors AB and A'B' are their respective flow vectors on each face. In this case, only the best displacement vector is kept. The vector CD is one that does not have a corresponding displacement vector on the other face and does not require additional processing.

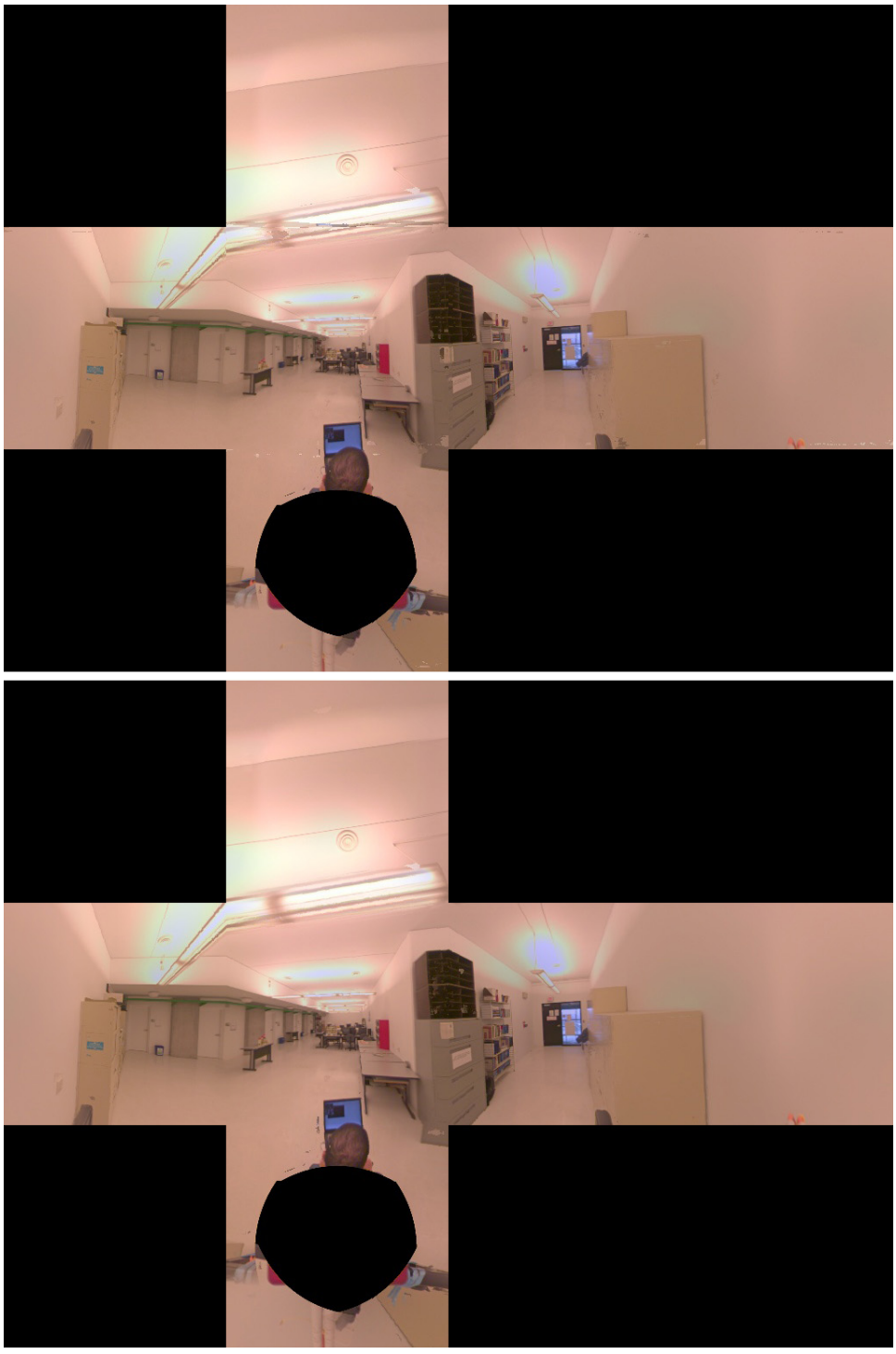

Figure 4.7: This figure shows an interpolated frame computed from a normal cubic panorama (top) and from an extended cubic panorama (bottom) at the same view position. Both are unfolded view of 256x256 face images. Some mistakes can be seen in the interpolation from normal cubes that are not present in the extended cubic representation (e.g. the fluorescent on the ceilling).

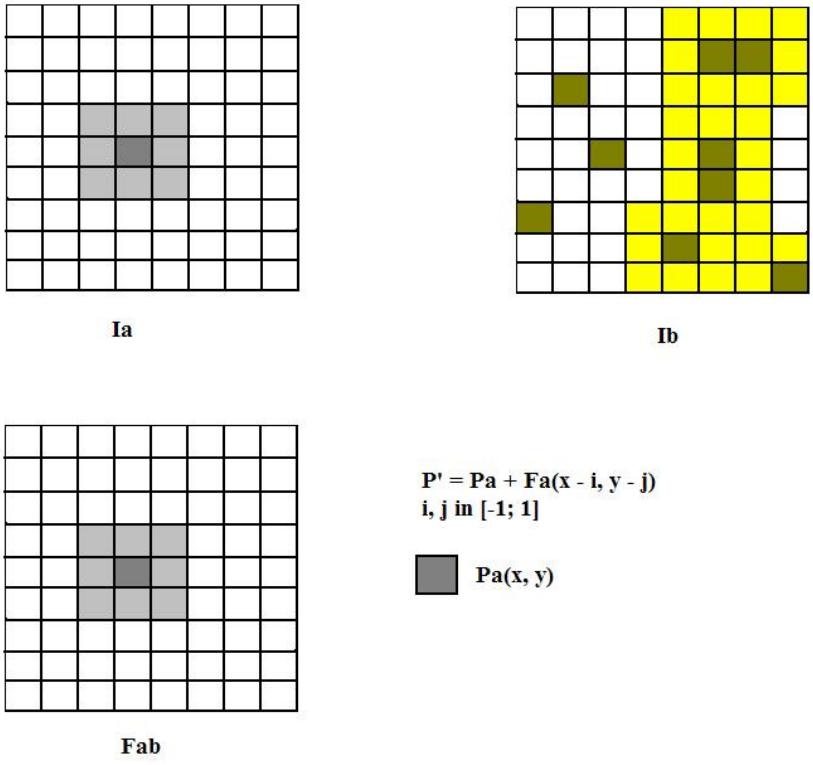

Figure 4.8: This figure demonstrates how we retrieve the replacement smoothing flow for pixel p_A.

4.2 Smoothing the flow vectors

Since our objective is to obtain realistic viewpoint transition, we observed that it is generally beneficial to smooth the optical flow field before using it for view interpolation. This additional step removes potential outliers that could introduce annoying visual artefact in the interpolated viewpoints.

As said earlier, we assume that no scene objects are moving in the scene and that there is only few abrupt depth discontinuities. It follows then that for each flow vector in the image, its neighbors should have a similar direction and orientation. Based on this observation, the smoothing step works as follows: for each pixel $o = (o_x, o_y)$ in the image I_1, we get its $n \times n$ color neighborhood N and its $m \times m$ displacement vectors neighborhood F. For each of these possible displacement value d in F, we calculate the displaced coordinate $o' = (o_x + d_x, o_y + d_y)$. We then get the $n \times n$ color neighborhood N'

of o' in I_2. We then compare N and N' using the L2 norm. If N' gives us the best possible match for pixel o in I_2, we select its corresponding displacement vector d as the actual displacement vector for o. To take into account a possible different displacement for o compared to p, we check the candidate displacement d with a certain offset. So instead of having $o' = (o_x + d_x, o_y + d_y)$ we actually use $o' = (o_x + d_x + offset_x, o_y + d_y + offset_y)$, where $offset_x$ ranges from $[i, j]$ and $offset_y$ ranges from $[k, l]$, where $i \leq j$, $k \leq l$, $i, k \leq 0$, $j, l \geq 0$. In Figure 4.8, we illustrate how to choose the possible replacement candidate for a certain pixel p_A of image I_A. For this pixel we retrieve its flow value and the one of its neighbors from image F_A. We get the possible replacement candidates from image I_B by applying the previously explained method, and we will compare the neighborhood of p_A in image I_A with the neighborhoods of all the black pixels of images I_B.

We summarized this step with the following pseudocode, using the same notations as previously:

```
Get N, the nxn color neighborhood of o in I1
forall the p in N do
    Find corresponding displacement d of p
    for offset_x = i to j do
        for offset_y = k to l do
            Get displaced pixel p' in I2, p' = (o_x + d_x + offset_x, o_y + d_y + offset_y)
            Get N', nxn color neighborhood of p' in I2
            comp = Compare N and N' using the L2 norm
            if comp is a better match than current value then
                use the current displacement value as the displacement for the
                current pixel
            end
        end
    end
end
```

The goal of this step is to smooth the flow field, in order to remove "rogue" vectors that have been mismatched or/and are oriented in a completely different direction than their neighbors. Chapter 5 will show the interpolation artefacts that result from these vectors and how this smoothing algorithm improve the results. These sparse outliers in the displacement field can cause very annoying effects. Figure 4.9 presents the un-

corrected flow and the smoothed flow of the same pair of panoramas, where the flow is represented as red arrows (which size has been enhanced to make the image more readable). When comparing these images, we observe that some rogue vectors have been removed from the flow field.

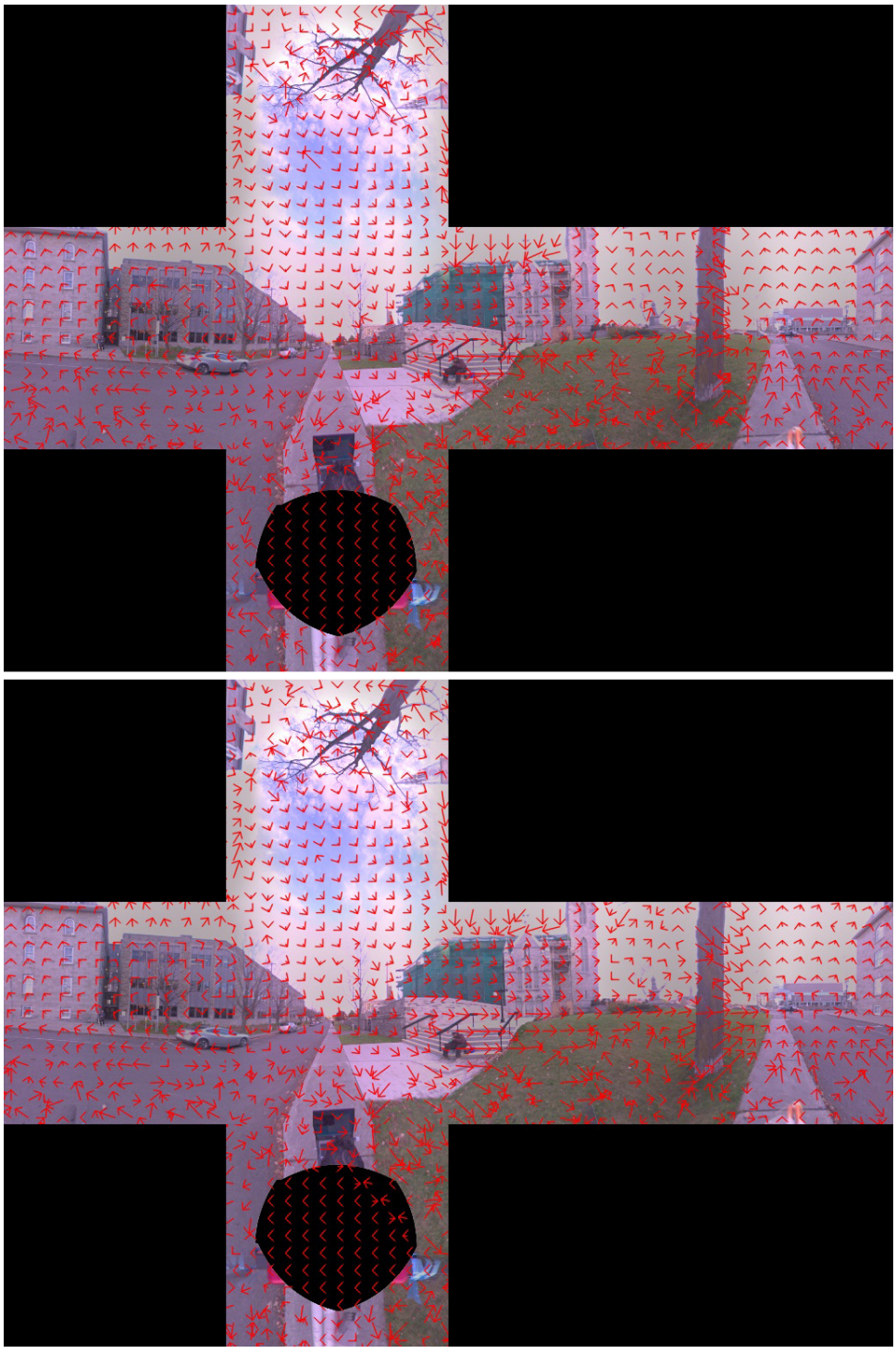

Figure 4.9: This figure illustrates the uncorrected flow that we retrieve from the optical flow algorithm (top) and the optical flow after smoothing it using our scheme (bottom). We draw the arrows for certain neighborhoods so the improvements of the optical flow can be seen more clearly.

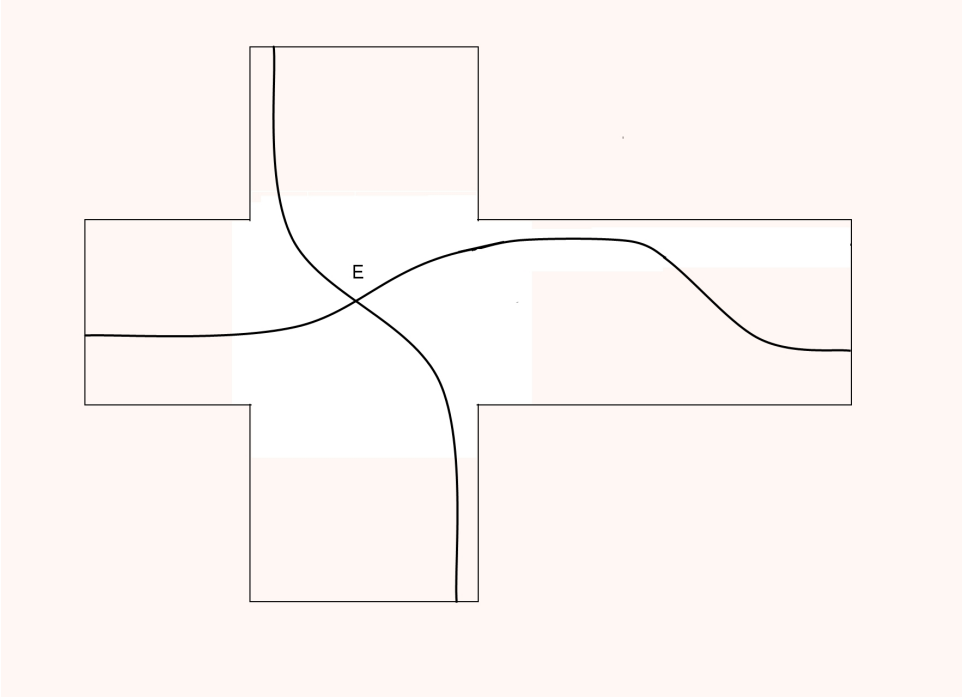

Figure 4.10: Illustration of the epipolar geometry on a cubic panorama.

4.3 Epipolar Geometry Correction

We tried two different ways of using the epipolar geometry of the scene to correct the optical flow. The first one was to reproject every displacement vector of our dense optical flow onto the epipolar lines obtained by calculating the epipolar geometry between the panoramas. The second was to constraint the optical flow algorithm so that for every pixel in the image, it would only search for flow vectors that were colinear that same estimation of the flow field. First we will explain how to estimate the epipole position from a pair of cubic panoamas. Next we will explain how we estimate the flow vectors for each face of our cubic panorama, and finally we will describe in details both methods that we mentioned previously.

4.3.1 Epipolar Geometry Estimation

Given that we are using 360° panoramic images of a static scene (i.e. not containing moving elements), the epipolar geometry can then be used to constraint the flow field computation between each pair of reference images. The epipolar geometry of a cubic panorama is illustrated in Figure 4.10, where E is the epipole, and the lines represent selected epipolar lines to which the flow vector of each pixel on that line should, in the case of static scenes, be colinear to.

When several panoramas are analyzed, they are linked by the usual projective relations. By definition the essential matrix is :

$$E = [t]_{\times} R \tag{4.1}$$

Up to a scale, E characterizes completely the geometry between two panoramas, just as in the case of conventional stereo images. It follows the following constraints for two matching points p and p':

$$p'^{T} E p = 0 \tag{4.2}$$

Since we know the calibration information of our camera, p and p' are expressed as 3D coordinates of the image points on the 3D projection surface (a cube in our case). Estimating E then becomes a matter of solving the classical problem of the epipolar geometry estimation using for example the 8-point algorithm [11]. Indeed, standard projective geometry applies to spherical pin-hole cameras in which the 3D coordinates of the image points on the reprojection surface are used in the homogenous equations.

Camera geometry estimation, as explained above, requires the matching of interest points across panoramas. We used SURF scale-invariant features and descriptors [4] to search for matching pixels among pairs of images. In order to reject outliers, a RANSAC scheme is applied to eliminate matches that are located too far from the epipolar plane defined by the currently estimated E matrix. Additional multi-view criteria, similar to [48], are then used to prune out the surviving false matches from the global match set.

4.3.2 Estimation of the flow fields on a cubic panorama

The epipolar geometry estimation gives us the 3D coordinates of the epipole. We know that every displacement vector in an image points toward its epipole. First we need to project this point on the panorama structure that we are using, either cubic, cylindrical or spherical to retrieve its coordinate on in texture. In our case, we are using a cubic image, with 6 images for each face, which means that we will have to project the epipole on each of the faces in order to be able to estimate correctly the vector field for each

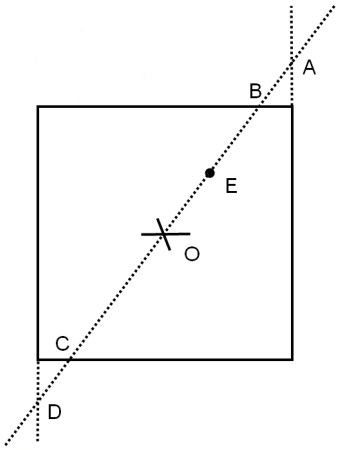

Figure 4.11: Illustration of the reprojection of the 3D epipole point onto a cubic panorama. (Illustration is seen from a top view). E is the 3D epipole point, O is the center of our cubic panorama, and A, B, C and D are the projections of E on the different faces of the cube.

face (see Figure 4.11). Once we have these reprojection coordinates, estimating the flow is done on each face independently. This is a straightforward operation, and only needs the application of the following formula:

$$EF(x, y) = p(x, y) - e(x, y) \tag{4.3}$$

Where EF(x,y) is the estimated flow vector for pixel (x, y) and $p(x, y)$ and $e(x, y)$ are the coordinates of the current pixel and the coordinates of the epipole on the face respectively. An example result for a single can be seen in Figure 4.12, where E is the epipole and the arrows are the estimated flow vector for a selection of pixels.

4.3.3 Reprojecting the Optical Flow

Now that we have an estimation of the dense flow field, there are a few ways that we can use it to enhance the quality of the resulting displacement field. In this section, we will describe the first improvement we tried which consists in reprojecting each calculated optical flow field vector onto its corresponding flow field estimate to give it the desired direction. We have decided to study this possibility because it would allow us to continue

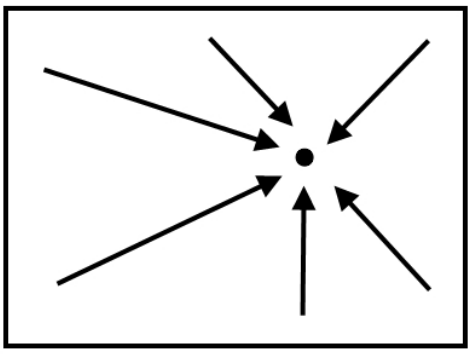

Figure 4.12: Illustration of the flow vectors for a single face of the cube.

being able to use any optical flow calculation algorithm transparently and providing a correction scheme to the optical flow that could be applied to any optical flow calculation algorithm.

Since we are estimating the position of the epipole by matching 2 panoramas, the accuracy of the placement of the epipole might not be perfect, and thus the results might not be accurate at first. That is why we have decided to run this correction step with the smoothing pass of Section 4.2, instead of having to choose one or the other. In our tests, we tried running the correction step followed by a smoothing step, and a smoothing step followed by a correction step and finally a smoothing step followed by a correction step and another smoothing step. We found that the results in test case 2 and 3 were very similar and of better quality thant he results in test case 1. The need for an accurate placement of the epipole is especially important when the epipole projection on the plane is inside or close to the face we consider. In this case a variation of the epipole coordinates might lead to big changes in the displacement field estimation. This applies to the face towards which we are moving, and to the face from which we are moving away. If the epipole projection coordinates place the point far outside of the face, then small changes in the position of the epipole will hardly affect the displacement field estimation. This is the case for the side, top and down faces.

A problem with this approach, for which we will see the interpolation result in the

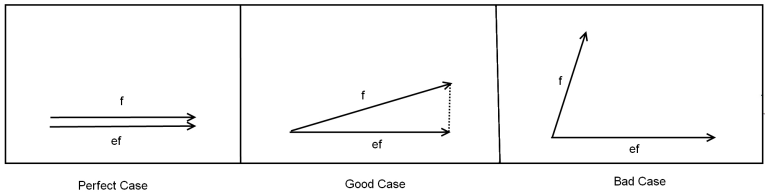

Figure 4.13: Illustration of the problems that can happen when reprojecting the calculated flow field onto the estimated flow field. The reprojection might be too long or too short and will loose the information obtained while calculating the optical flow which leads to different artefacts in the result image, such as blurring.

next Chapter, is that in all the cases that were tested, the few improvements in quality that might have been obtained through its use did not outweight the reduction of quality of the scene. The best explanation for such result is that when we reproject the flow onto the epipolar estimation, we change the length of the vector, which changes the matching that occured during the optical flow estimation (See Figure 4.13 for an example). In Figure 4.14 we display the reprojected flow field of one of our pair of panoramas.

Thus we have devised a second scheme, which consists of constraining the optical flow algorithm to only search for a possible shift which is colinear to the expected flow field. We will describe our approach in the following section.

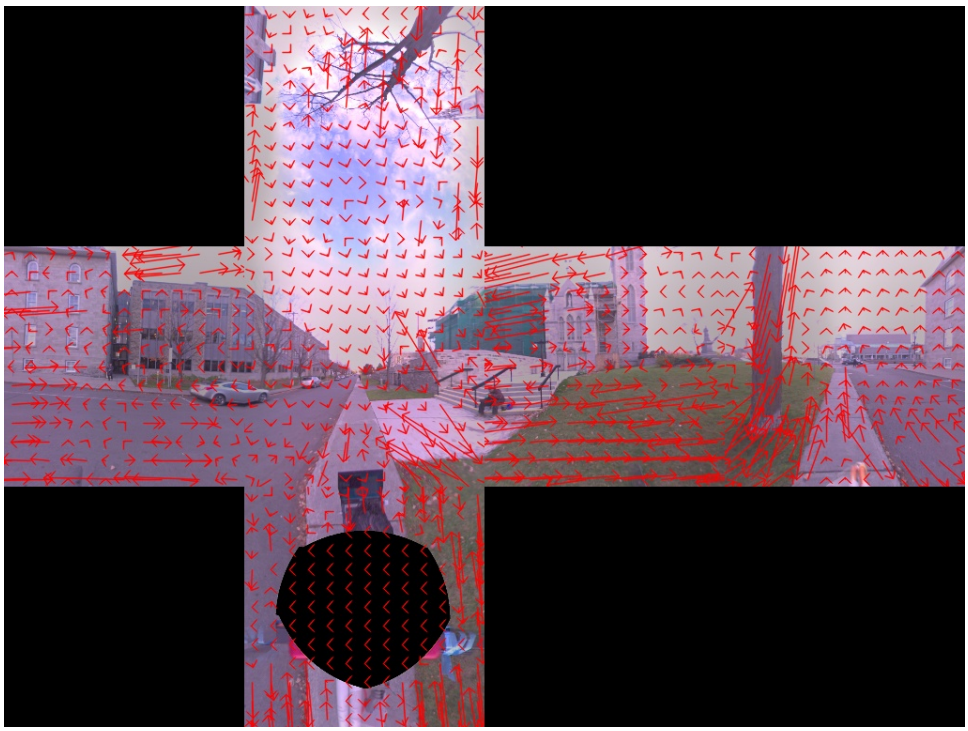

Figure 4.14: This figure illustrates the reprojection correction step applied to the flow.

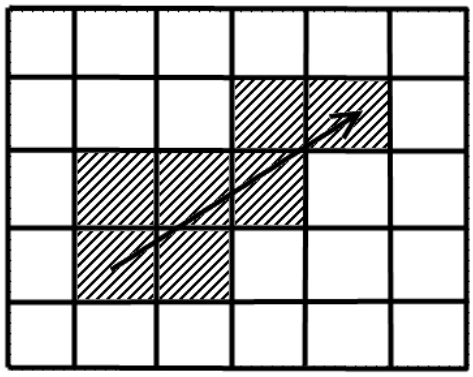

Figure 4.15: For a given estimated flow vector (arrow), the flow vectors that are accepted by our constraining scheme are the one that have the same starting pixel as the estimated flow vector and one of the colors pixel as their endpoint.

4.3.4 Constraining the Optical Flow Algorithm

The second idea that we experimented with consisted in constraining the optical flow algorithm to only consider the candidate shifts that were colinear to the estimated epipolar lines. The idea of constraining the possible optical flow vectors to those determined by the epipolar geometry estimation is a fairly straighforward modification. For each pixel in the image, we will not consider a shift vector that is not similar to the estimated epipolar vector. As we mentioned with the previous method, the epipole might not be entirely accurate, and thus we allow the candidate displacement pixel to be a few pixels offseted from the estimated flow vector. In Figure 4.15, the arrow shows the estimated flow vector at its maximum length for a certain pixel, and the black pixels show all possible candidates that we consider. The maximum length for the vector is determined by the user selected [minX, maxX] and [minY, maxY] as required by the optical flow algorithm.

Instead, we first set our estimated epipolar flow vector to have the same length as the shift vector. Then we say that the distance between the endpoints of both vectors

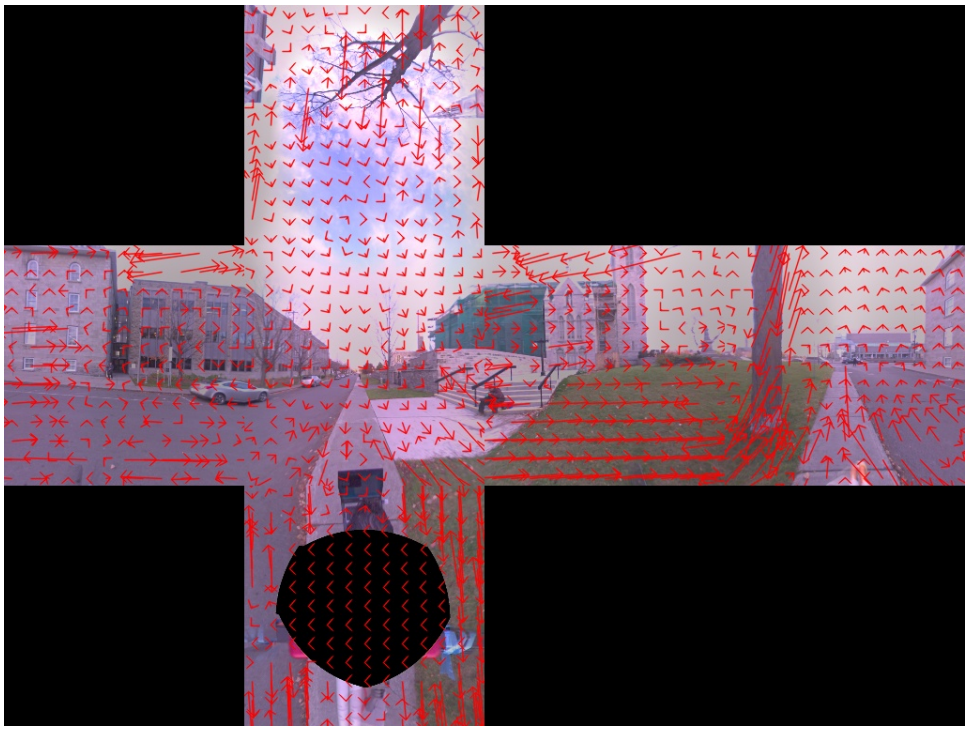

Figure 4.16: This figure illustrates the constraining of the flow followed by a smoothing correction as described in this section.

applied to the same pixel, should be less than a user defined number of pixels (in our case 2). This is similar to saying that the shift should reach at least a pixel close to 2 pixel to the estimated epipolar vector. If this distance is less than a predefined number of pixel, we accept it, if not we ignore it and continue to the next possible shift until the last one is reached and a best candidate has been found. Figure 5.6 shows a result of this constraining scheme on one of our pair of panoramas.

4.4 Conclusion

In this Chapter we have presented a way to extended the current optical flow algorithm to work with cubic panoramas, and we have experimented with various correction schemes that can be applied to any optical flow in order to make the interpolation results more immersive. The smoothing flow increases the result in most cases. We also experimented with using an estimation of the epipolar geometry in order to constrain the optical flow calculations, and even though the results are not better at this state, in certain parts

of the image it improves the quality of the results and shows good promises for future research. In the next Chapter we will describe the interpolation process using the calculated optical flows between pairs of panoramas and we will present some interpolation results of the different correction steps that we presented in this Chapter and a few discussions on the different results that we obtained.

Chapter 5

View Interpolation

Our view interpolation is based on the algorithm developed in [32]. This algorithm was designed to work on planar textures in order to create new textures by combining already existing ones taken from a given database. Using this approach, textures can be morphed by moving along a defined visual path.

In the first section we will describe the parts of the algorithm that are relevant to our approach. In the second section, we will present our approach to view interpolation on real-life images.

5.1 Basic algorithm description

Th method in [32] was designed to create realistic transition textures between multiple input textures. A wide variety of input textures can be used, from grass to brick wall textures. There are multiple important components to this method, but the ones of interest to us and the ones that we will describe in the following subsections are the feature extraction, the warp computation and the morphable interpolation.

5.1.1 Feature Extraction and Warp Computation

The first thing the algorithm does is to extract features in each of the input textures. To do so, the authors have selected the compass operator introduced in [39], which extracts oriented edge features from the texture samples. This operator returns the orientation and strength of the maximum reponse at each pixel inside of a gray scale feature map texture, for 2 textures F_i and F_j. Once the feature map of every texture has been calculated, they are used to compute the warping between textures. This warping

function W_{ij} defines, for all pixels $p(x, y)$ in F_i a corresponding pixel $q(x, y)$ in F_j such that $q(x, y) = p(x, y) + W_{ij}(x, y)$. Their algorithm uses a pyramidal approach to compute a dense displacement map. The pyramid goes from fine to coarse or from the lowest to highest level of the pyramid respectively.

- The lowest level of the pyramid is set to be of the size 16x16. Feature maps are overlaid onto the texture using with a regular 8x8 triangulation (each vertex is connected to 6 of its neighbors).
- Next the triangulation of F_i is kept fixed while the vertices F_j are modified to find an optimal match for each of these vertices in the triangulation by trying to warp them into the triangulation of F_i. This step effectively calculates the warping function W_{ij} for every pixel in the image.

5.1.2 Morphable Interpolation

The most important feature of this method is the one regarding the calculation of the morphing between textures. The method is based on [33] where the warp functions are calculated with respect to a global reference texture which is invariant for every texture in the database. In contrast this method calculates the warp between pairs of images. Each input texture is assigned a weight, with the constraint that $\sum_j w_j = 1$, and a barycentric morphing between n textures I_i with $0 \leq i < n$ is applied to create a new interpolated texture. There are 2 steps to this algorithm: the linear morphing of color and the linear morphing of the geometry of the texture. This is done using the following equation:

$$\hat{I}(x, y) = \sum_{i=0}^{n-1} c_i I_i((x, y) + \sum_{j \neq i} w_j W_{ij}(x, y)^{-1}) \tag{5.1}$$

Where $\hat{I}$ is the morphed texture, and c_i and w_j are the weights of the color and shape interpolation respectively. The constraint $\sum_j w_j = 1$ is also used to ensure that the textures are properly aligned. When setting $w_j = 0$ for all possible j this corresponds to linear blending.

5.2 Interpolation on Pairs of Real Images

Our algorithm is designed to work on cubic panoramas, on a per face basis. We are using real-life images, that are taken close to each other, in a given city/environment. The first step of our algorithm is to compute the dense flow field between both images using the method described in the previous chapters. Once this dense correspondence between all pixels in both panoramas has been established, we can proceed to interpolation at intermediate viewpoints. The straightforward approach would be to use linear blending. The problems with this type of blending is that it is unrealistic and does not take into account the geometry of the scene and the motion between the views. Our main objective is to make the transition between views as realistic as possible. This way the virtual view navigation will become a continuous walkthrough inside the scene. We alleviate these shortcomings by including the dense displacement maps into our morphing algorithm. Instead of only interpolating the colors of the scenes, we also use the displacement vectors in the interpolation. Equation (5.1) can be written for pairs of images ($n = 2$) as follows:

$$\hat{I}(x,y) = (1-c)I_0(wW_{01}(x,y)^{-1}) + cI_1((1-w)W_{10}(x,y)^{-1}) \tag{5.2}$$

with the following definitions:

- $\hat{I}$ is the transition image
- I_0 and I_1 are the origin and destination images respectively
- W_{ij} is the dense displacement field from image I_i to I_j. The optical flow algorithm that we use makes the assumption that $W_{01}(x,y) = -W_{10}(x,y)$
- c and w are the weights applied to the color and displacement respectively, and depend on our position between both images. In our experiments, we set $w = c$, because we want the color and the geometry of the scene to be interpolated jointly.

In Figures 5.2 and 5.3 we show in each of them the same transition image interpolated with two different schemes, one taken from an indoors sequence, the other from an outdoors sequence. In each of the figures, the top one was obtained through linear interpolation and the bottom one was obtained using the method we just described. There are still a few artifacts in the interpolated image mainly in the capturer's head and the fluorescent on the ceiling. However, the image still looks sharp and clear whereas the linearly interpolated version looks blurry and has many artefacts. There is, in the authors' opinion, clear improvements in the quality of the image and immersion of the user when using our version of the transition images.

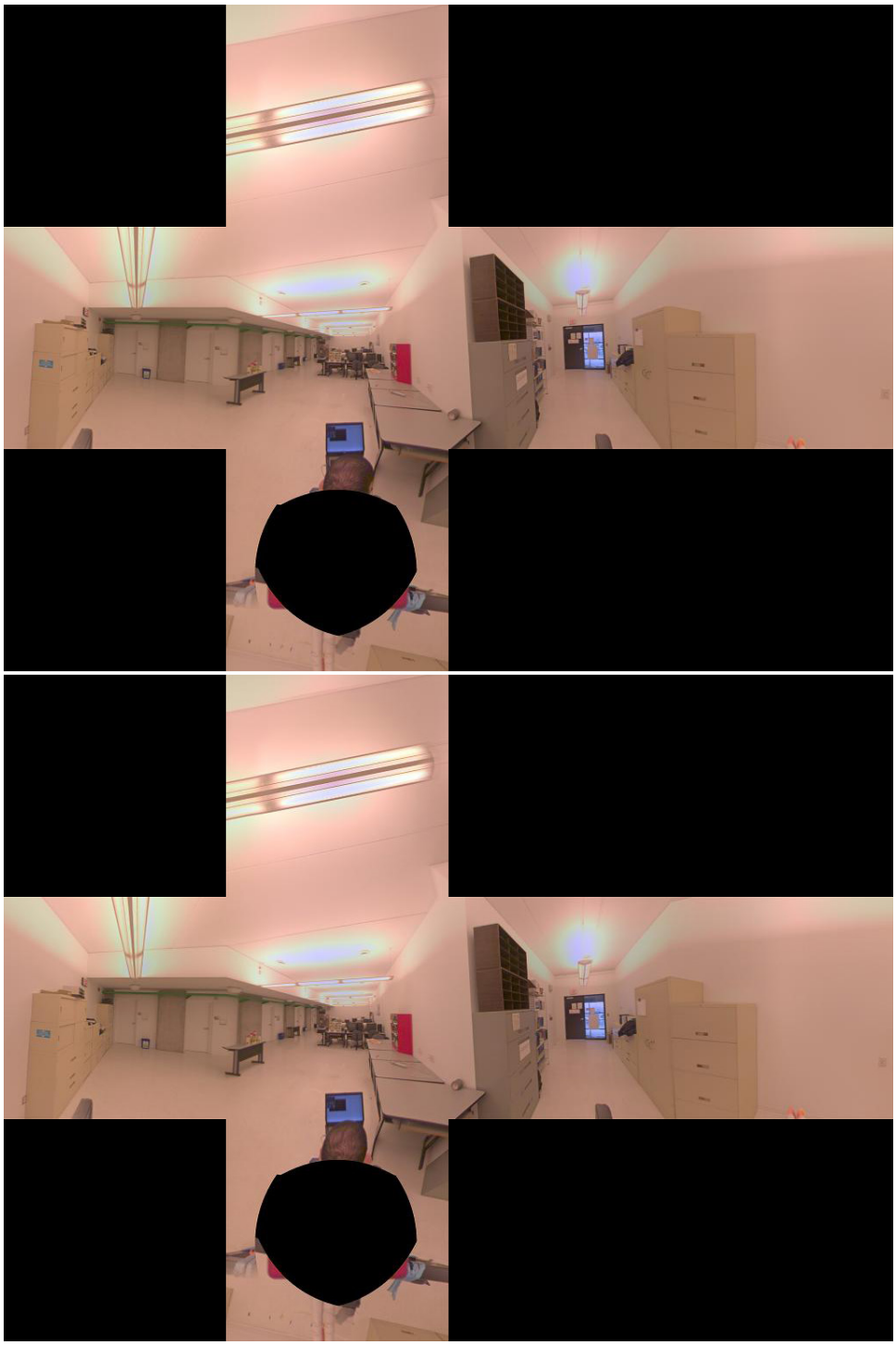

Figure 5.1: This Figure shows the origin and target pairs used to do the interpolation of the images shown in Figure 5.2

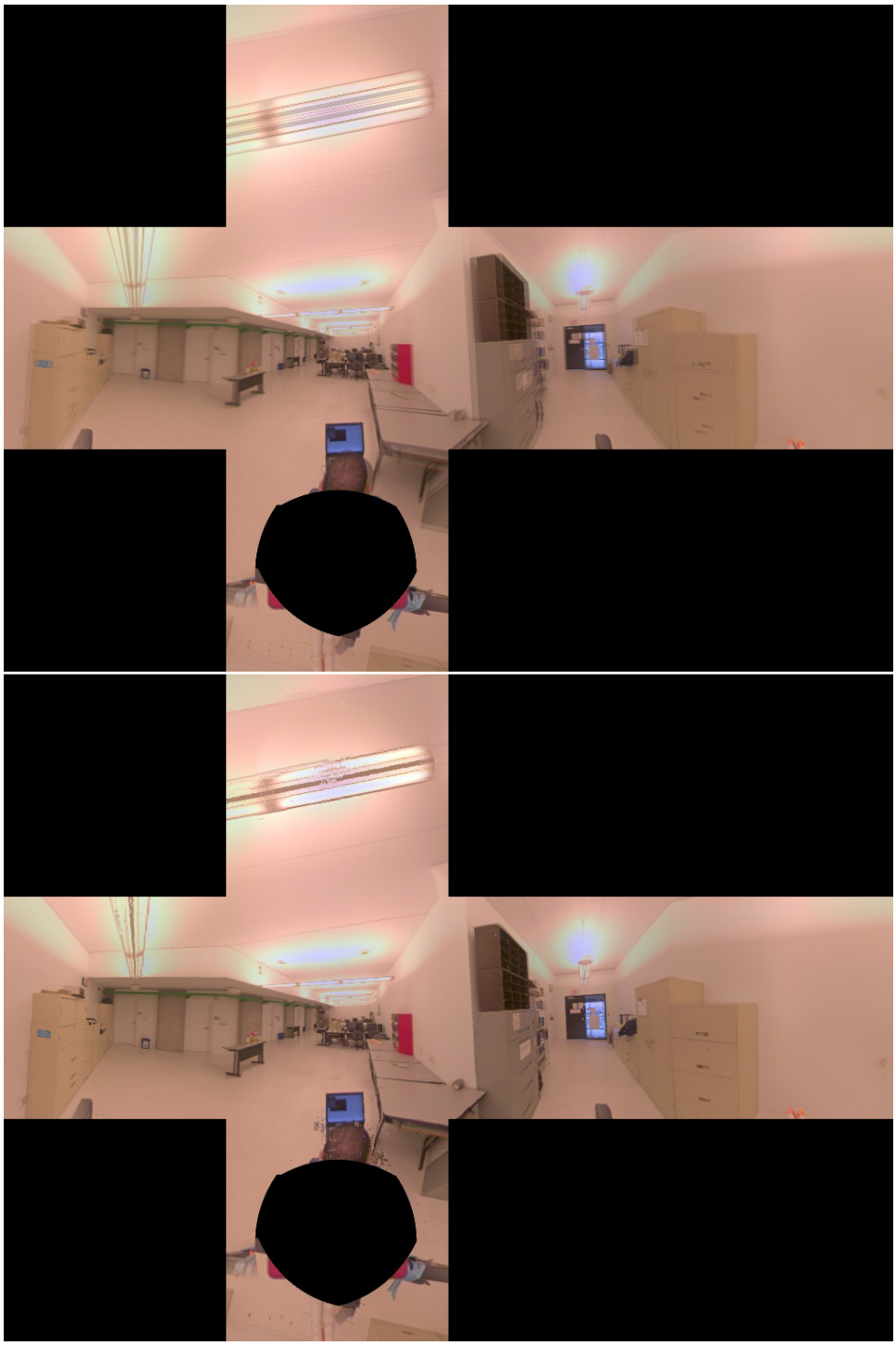

Figure 5.2: This shows 2 different interpolation: the top one using a linear interpolation scheme and the bottom one the interpolation method presented in Section 5.2. Both are interpolated from the same origin and target images that are shown in Figure 5.1, and using the weigths c = w = 0.5

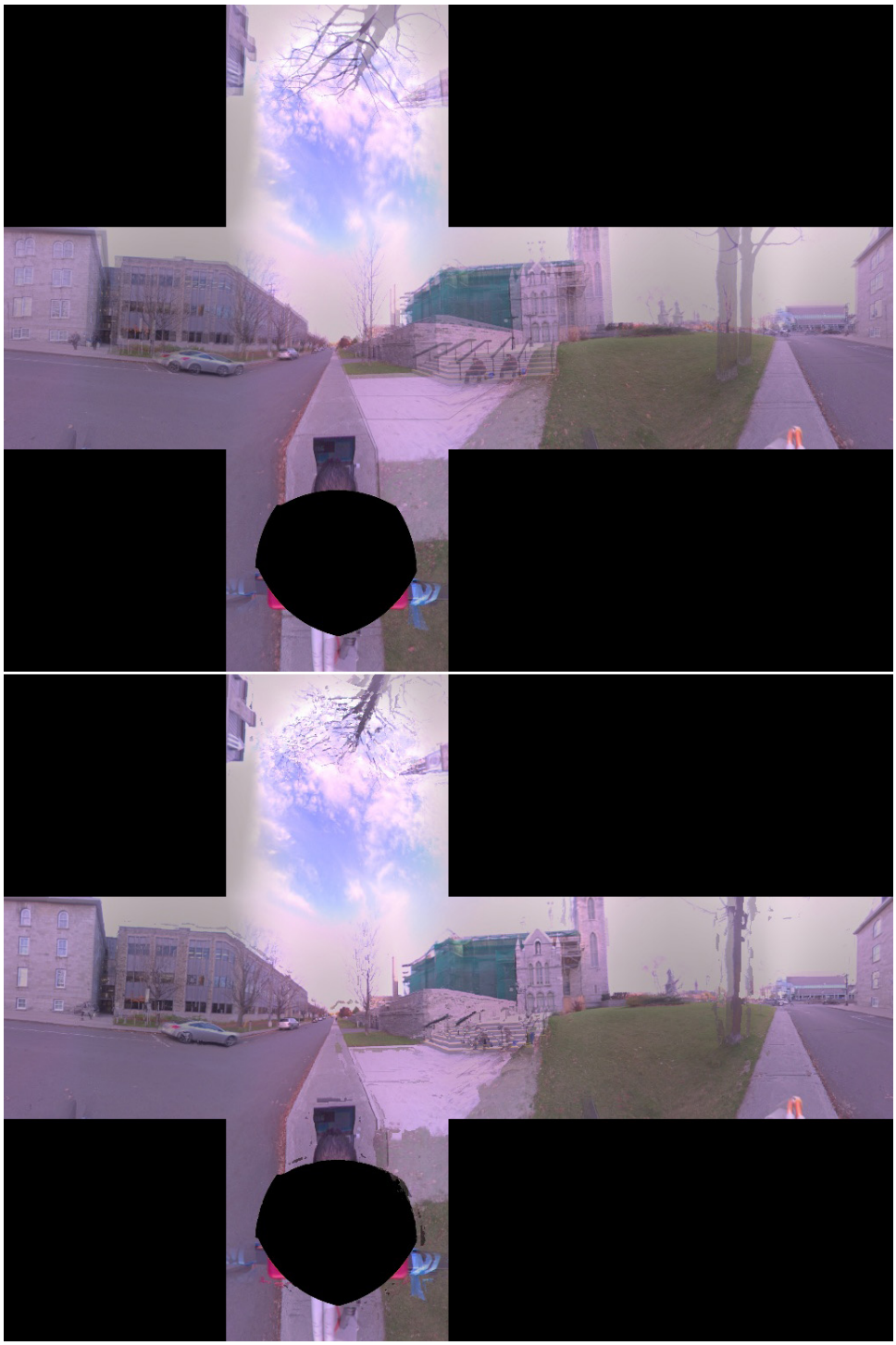

Figure 5.3: This shows 2 different interpolation: the top one using a linear interpolation scheme and the bottom one the interpolation method presented in Section 5.2. Both are interpolated from the same origin and target images that are shown in Figure 5.1, and using the weigths c = w = 0.5

5.3 Results using the various flow correction schemes

To finish we will show a couple of interpolation results with the different kind of corrected flow that we introduced in the previous chapter. In Figure 5.4, we show the same part of different panoramas to illustrate the improvement brought forth by using the smoothing correction. The improvements are obvious in the interpolated image using the smooth flow compared the image obtained when using the uncorrected flow: the image is sharper, the plant kept his shape. The improvements are also noticeable around the posters on the wall. The static image already shows some noticeable improvements, but using the smoothing correction becomes even more worth it when moving in the environment.

In Figure 5.5 we show an example of such a result. To obtain the top image, we used our scheme with only the smoothing correction step, and to obtain the bottom image we used our scheme using the epipolar reprojection followed by a pass of the smoothing correction. The quality of the results are much lower when using the epipolar reprojection correction and thus, as we said previously, this is not a usable correction.

Figure 5.6 shows a result of this constraining scheme on one of our pair of panoramas. We see from this result that even though the result of this constraining step is overall less good that the one obtained by only applying the smoothing step, some part of the image are better and it will be an area of interest for future work.

5.4 Conclusion

In this Chapter, we have shown how to interpolate between pairs of panoramas. In the next Chapters we will explain how to achieve real-time navigation using all the elements presented in the previous Chapters and present some results of our method on different captured sequences.

Figure 5.4: The top images are a part of 2 captured panoramas. On the left the origin image and on the right the target image. The bottom images are interpolated images with interpolation coefficient 0.5. On the left the interpolation with a smoothed flow and on the right we applied without smoothing. We can see that without smoothing the plant looks blurry and has many artefacts, whereas it keeps its sharpness and shape when using the smoothing step.

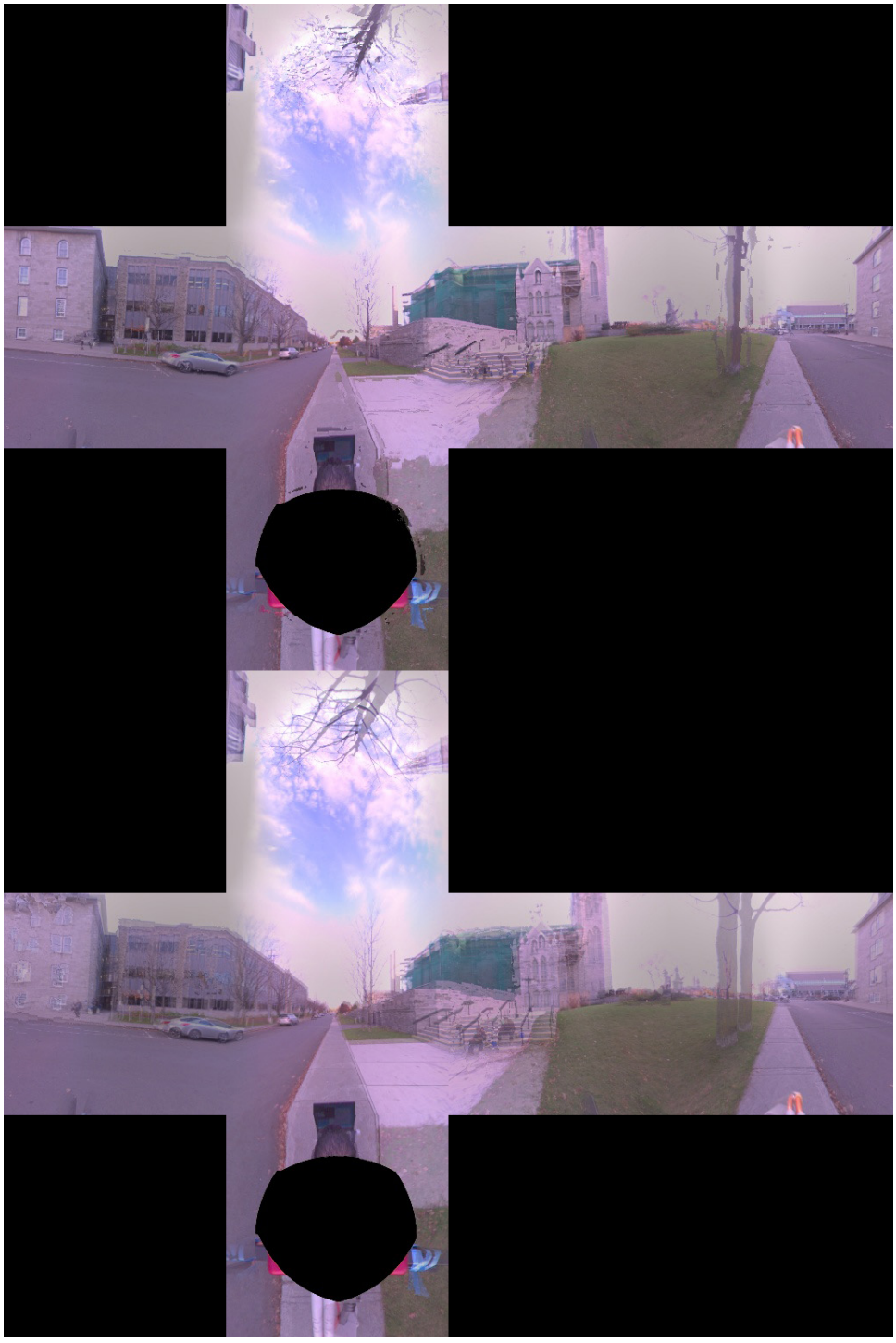

Figure 5.5: This Figure shows a middle transition image using the smoothing correction flow only (top) and the epipolar reprojection and smoothing correction (bottom).

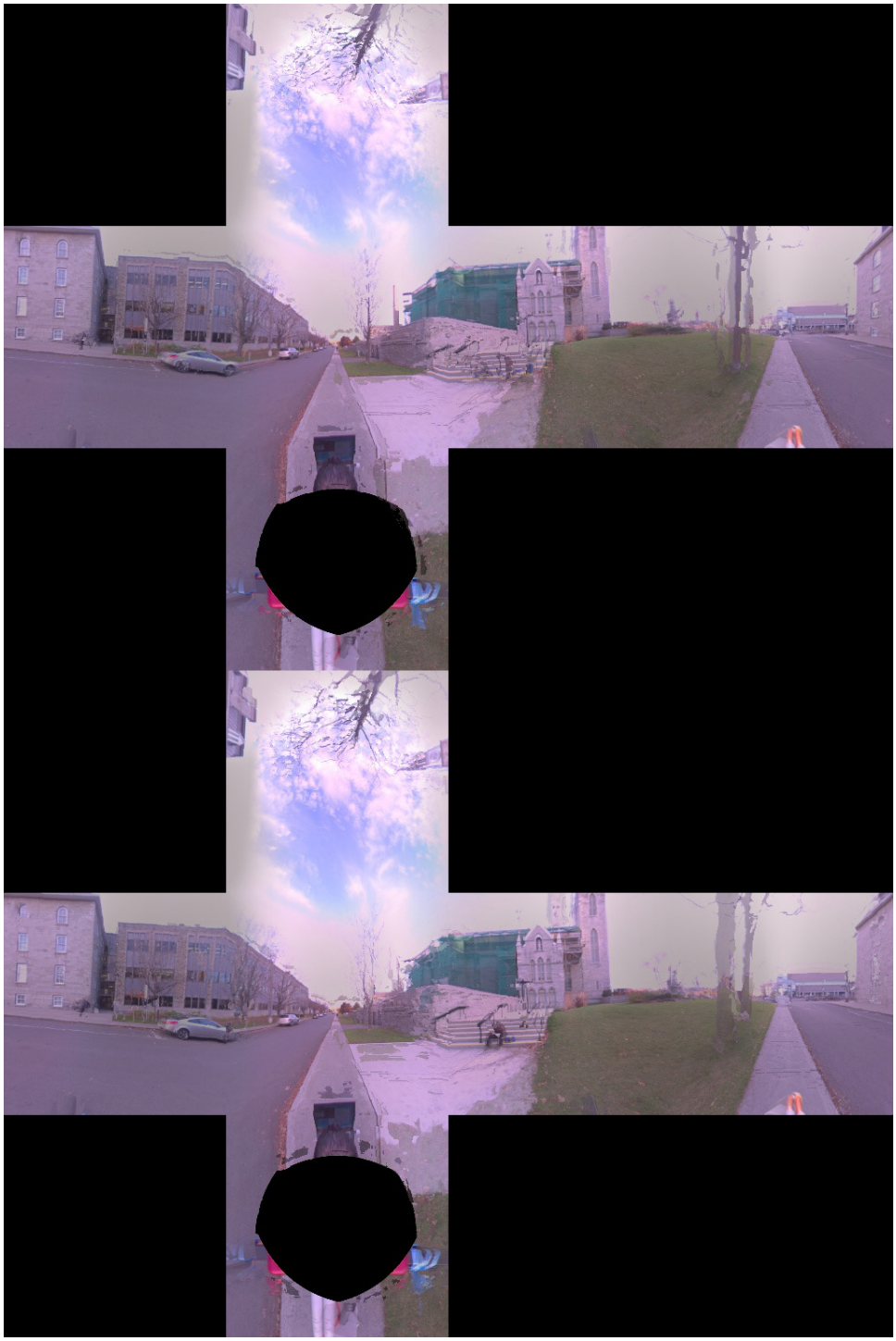

Figure 5.6: On the top is an interpolated image (with w=c=0.5) using the smoothing correction, and on the bottom is the corresponding image when using the epipolar constraining of the optical flow and the smoothing step.

Chapter 6

Real-Time Implementation

Real-time intermediate view generation requires the preprocessing of the optical flow, data buffering, multi-threading and the implementation of the interpolation algorithm on the GPU. We show in this section how these are implemented to achieve our goal.

The computation of the optical flow is the most time consuming step but as it can be precalculated, its estimation time does not affect the rendering performance. During navigation, the reference images and the corresponding computed optical flow field are loaded into memory. Its implementation does not require any specific treatment and thus we will only focus on describing the buffering of the data, the multi-threading of the application and GPU implementation in this chapter.

6.1 Buffering

Since we have many viewpoints for each scene, we will need to access the hard drive hundreds of times when navigating the scene in order to retrieve the different panoramas and displacement fields. These operations will greatly slow down the application because accessing the hard drive is one of the most time consuming operation to be undertaken on a computer. Therefore, we need to define an efficient strategy to retrieve that data in order to minimize this loss of time. To do so, we have decided to buffer the required data that will be the most likely to be accessed within the next few steps of our navigation. If we are currently viewing panorama C_i, then we are going to load the n closest panoramas to C_i using a breath first search: that is we will first get the neighbors that would require one hop to get to from C_i, and if our buffer size permits, we load the ones requiring two then three steps and so on until our buffer is full. As soon as the panorama viewed

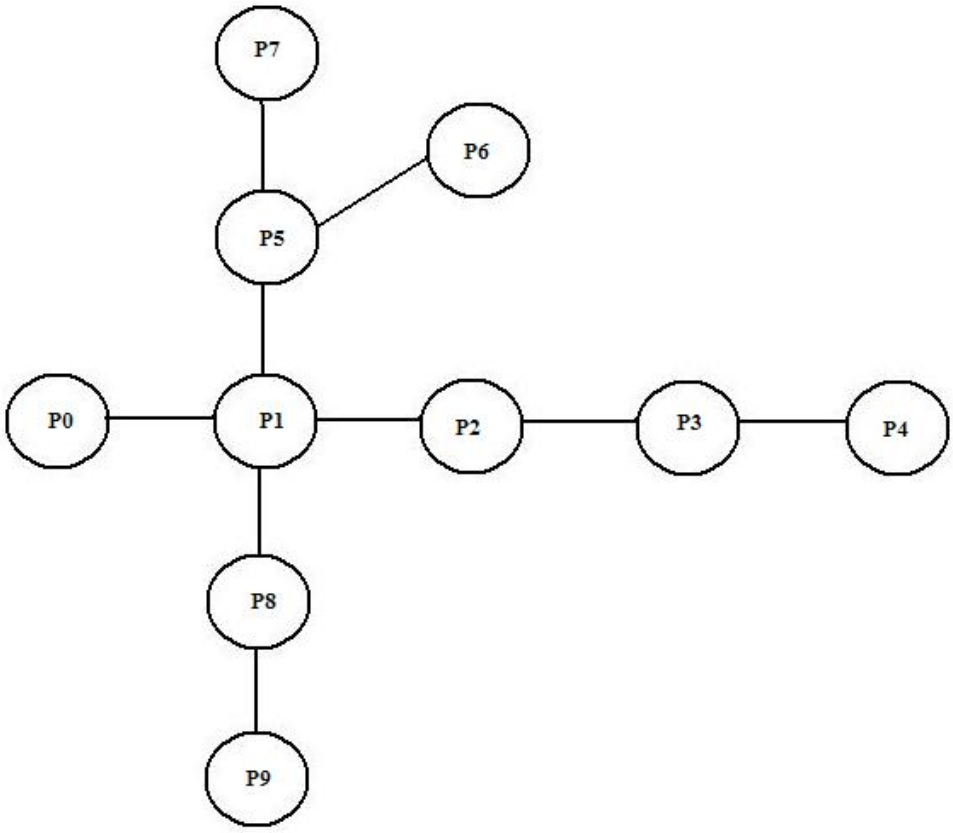

Figure 6.1: This graph is an example of a possible panorama setting. Let us suppose that our buffer size is 7. If the user is currently at the viewpoint P1, then the buffer is filled with the panoramas and displacement fields of the following viewpoints: P1, P0, P2, P5, P8, P3, P6. If the user is currently at P1, then the buffer is filled with the data of the viewpoints: P2, P1, P3, P8, P6, P5, P4

changes to C_j, we will need to check which panoramas are now too far from C_j, and which ones are now closer and need to be loaded, so that we release them and load them to memory respectively. The same process applies to the displacement fields. (See Figure 6.1 for an example)

6.2 Multi-Threading

For the buffering of the necessary data to be efficient and useful, we need the application to be multi-threaded. We use one thread to handle the graphics calls (in our architecture, the OpenGL calls), and another one to load the data from the hard drive, and finally the

last thread, which is actually run on the GPU, to calculate the interpolated images. If we had not used multi-threading, and simply used a single CPU thread, we would have to load the images and the displacement field each time we change images. This would slow down our application tremendously, since we would need to wait for both access to the hard drive (to get the target image and the displacement field) to be finished before being able to calculate and display the intermediate images. Using a multi-threaded approach allows us to constantly have a thread in the background checking for the necessary data and loading it, without the user noticing any downtime. One drawback to using OpenGL is that it does not support multi-threading, thus we need to load the textures in one thread and bind them in another, which makes the code more complicated.

6.3 GPU Implementation

In this section we will first briefly describe the evolution of the different Graphic Programming Units (more commonly refered to as GPUs) and next we will describe the GPU implementation of our view morphing algorithm.

6.3.1 A Brief Explanation of GPU Programming

GPUs were introduced to handle graphic intensive calculations such as texturing and lighting of pixels. In the early years, only a fixed set of functions were allowed to function on the graphics card which only allowed the developpers to use the features implemented by the graphics card manufacturer. As of February 2001, programmable vertex shaders and later pixel shaders were introduced to the GPU and allowed the developpers to implement their own way of handling the previously hardwired functions. Each shader could now be developped specifically for a certain software and the developper was in control of what the graphics card was doing. The last improvement to GPUs are the introduction of General Programming GPUs (GPGPUs) which allow the developper to not only program the GPU to handle graphics calculations, but also more general operations such as physics calculations and artificial intelligence calculations amongst other through the use of new languages: CUDA or OpenCL. GPUs have long been able to carry out a much higher number of operations than the CPU, and the difference is still steadily growing. According to [1], the Peak in GFLOP/s and the Bandwidth in GB/s for the latest GPU as of 2008 was 10 times higher than the latest CPU. This very high difference in performance is partly due to the fact that GPUs are aimed to handle only

graphics calculations whereas CPUs are aimed to handle a wider variety of tasks, but it is also due to the fact that GPUs are aimed to handle parallel computations in their architectures. To handle these graphics, parallel calculations, the GPU devotes more transistors to Data Processing, and has a much higher number of Arithmetic Logic Unit (ALU) than the CPU and much less cache and control transistors for storing information. The downpoint of this approach is that GPUs are not able to cache much data, but this is more than balanced by the fact that the calculations are much faster, and as such the caching of the data is replaced by the calculation of said data when necessary. The programs ran by the GPUs (shaders) are ran on each pixel of an image and should be calculated independently between one another, because it is impossible to know in which order the pixels will be treated. This makes it a strength of the GPU and a challenge to the developper. To learn more about the history of GPUs and their architecture, the reader is encouraged to read [29].

6.3.2 View Interpolation Implementation

The GPU implementation is the last part of our processing chain. Since we have already precalculated the flow, the interpolation of each pixel value is independent from the rest of the image which makes it a perfect candidate for parallel implementation on the GPU. The GPUs on the other hand are much faster than CPUs and are designed to handle intensive graphics calculations, and allows us to do these calculations while the CPU is handling other kinds of operations (graphics calls and hard drive calls in our case). In addition to the panoramas and the displacement fields needed by the interpolation, we also need to pass the optical flow window boundaries that we used in the optical flow calculation algorithm as well as the texture sizes in order to be able to calculate the origin and target pixel coordinates of each of the interpolated image pixels directly on the GPU accurately. We describe the GPU code in the following pseudo-code, inspired from GLSL code:

uniform sampler2D: OriTex, TarTex, FFX, FFY;
uniform floats: fDispIC, fColorIC, fDispLgthX, fDispLgthY, fTexSz;
TEXCOORD: pixCoord
vec2 $iPixCoord = pixCoord * fTextureSize$;
vec2 $offset$ = vec2(0.0f, 0.0f);
$offset_x$ =(texture2D(FFX, pixCoord)-0.5);
$offset_x$ *= fDispLgthX;
$offset_y$ =(texture2D(FFY, pixCoord)-0.5);
$offset_y$ *= fDispLgthY;
vec2 $texcoordOrigin = (iPixCoord - fDispIC * offset)/fTexSz$;
vec2 $texcoordTarget = (iPixCoord + (1.0f - fDispIC) * offset)/fTexSz$;
//Interpolate the colors
$gl_FragColor = (1.0f - fColorIC) * texture2D(OriTex, texcoordOrigin) +$
$fColorIC * texture2D(TarTex, texcoordTarget)$;

First a few variables are declared. The first ones are uniform sampler2D are 2D textures which contain the origin (OriTex) and target (TarTex) textures color values, and the flow field information in both X (FFX) and Y (FFY) direction. The next set of variables are floating point values which indicate the displacement(fDispIC) and color(fColorIC) weighting factor which indicate where the user currently is between both images (0 meaning that the user is seeing the origin image, and 1 the target image). The next two variables (fDispLgthX, fDispLgthY) are the variables used by the optical flow algorithm. To make the algorithm easier to summarize here, we assume that the optical flow used a window of [-fDispLgthX/2; fDispLgthX/2] for x and [-fDispLgthY/2; fDispLgthY/2] for y. Finally fTexSz is the size of the texture, which we assume is square and pixCoord is the coordinate of the current pixel. The variable pixCoord is calculated on the GPU and is contained within the range [0, 1], where 0 is the leftmost side of the texture and 1 the rightmost side. For our method to work properly, we need to convert these values to it's integer coordinates which will be in the range [0, fTexSz], and this is what is done in Line 4. In Lines 6 and 7 (resp. 8 and 9) we retrieve the X (resp. Y) flow value, which is constrained within the range [0, 1] and convert it back to its original [-fDispLgthX/2, fDispLgthX/2] (resp. [-fDispLgthY/2, fDispLgthY/2]) range of values. In Lines 10 and 11 we calculate the original and destination coordinates of the current pixel using the displacement interpolation and convert it to be in the range [0, 1] so that we can use them to do the color interpolation in Line 12.

6.4 Conclusion

In this Chapter, we have presented the different elements to achieve interactive navigation of any environment captured through a series of panoramas. In the next Chapter we will present different results of our method and of the different algorithms presented in this thesis work.

Chapter 7

Results

In this chapter we will present the different results that we obtained during our experiences.

7.1 On the Data Acquision Times

We captured our scenes using a Ladybug camera, which uses 6 cameras of resolution 1024x768 to create panoramic images in a single shot (5 on the sides, and 1 at the top). We mounted the Ladybug camera on an electric scooter equipped with a computer and a GPS device. The images captured from the Ladybug are saved and converted to a cubic texture format. We are using extended cube faces of size 320x320 pixels each, while the normal cubes have faces of size 256x256.

The optical flow calculation and correction took up to an hour on a 320x320 image on an Athlon 64 X2 6000+/3GHz. We ran the optical flow calculation with an interval for both x and y of [-40; 40] and we set the size of all the neighborhoods in the correction pass to 11. The time needed to do these calculations is not constraining because these are done during a preprocessing stage.

To evaluate the computational load of the interpolation process, we ran another 2 rounds of calculations on the same set of panoramas of resolution 320x320, one using only the CPU and the other using our GPU implementation. To achieve a smooth transition between images, we need to calculate at least 20 interpolated frames between each pair of images. On the CPU, we created 20 transition images in 3 seconds. With our GPU based interpolation scheme, we could generate up to 1000 images in 3 seconds, with identical quality. Our GPU implementation only requires a graphics card supporting the shader

model 1.0 and higher. Since the arrival of Windows Vista, most computers now come with an on-board graphics chipset that supports shader programming which makes this approach is accessible to a wide range of computers. We ran our tests on a Pentium M 1.7Ghz with a Radeon Mobility X700; more recent computer will perform even better.

7.2 On the Interpolation Evaluation

It is difficult from only the origin and target image to assess the accuracy of the generated intermediate, especially regarding the position of the objects. In order to evaluate the quality of the interpolated images, we ran the following test: we captured a sequence of panoramas and then we calculated the flow between the odd panoramas only, ignoring the even panoramas, these ones being used for comparison purposes. We then evaluated the displacement field and interpolated between each one of them using our scheme. To compare our interpolated panoramas with the stored panoramas, we assumed that all our panoramas are at equal distance from each other (our scooter was moving at constant speed). Using this assumption, we tested a few of the interpolation weights around $c = 0.5$ and $w = 0.5$ and kept the value giving the best result to interpolate between I_n and I_{n+2}. Comparing the two gives a good estimate of the quality of the interpolation scheme. The results for one of our sequence are shown in Figure 7.2. It can be seen that the quality of the interpolated image's geometry is very similar to the corresponding reference panorama with few artifacts visible. One important point is that since this interpolation is built for real-time navigation and the user will not be stopping at an interpolation image, so the smallest of artifacts will go unnoticed. Obviously, improved optical flow algorithms would enhance the quality of these results.

In Figure 7.2, we show the interpolated images with the interpolation coefficient that best correspond to the real (ground truth) panorama captured at interpolation location. In Figure 7.3, we compare the interpolated extended cube panoramas with the ground truth image from the same sequence as Figure 7.2. We compare the images pixel by pixel to a reference panorama and display the resulting image in gray scale: the clearer the pixels in the image are, the more different the pixels are. For each of the images, we get a root mean square as a measure of comparison between the ground truth image with: the origin image; the target image; the linearly interpolated image; the interpolated image using uncorrected flow; and finally the interpolated image using the smoothed flow. These results are indexed in Table 7.1. These results show that our interpolated images are much similar to the actual image that the linearly interpolated image. The

RMS Values when comparing the ground truth with the mentioned image				
Origin	Destination	Linear	Uncorrected	Smoothed
18.4965	18.709	14.7515	9.67674	9.14825
25.5349	25.6267	20.6691	13.9703	13.6975
29.5894	31.5904	25.2227	15.8723	14.6142
27.7342	27.4573	22.0606	12.2229	11.2739
22.61	21.49	17.55	14.02	14.05

Table 7.1: Table showing some of our RMS results comparing a ground truth image with: the origin image; the target image; the linearly interpolated image; the interpolated image using uncorrected flow; and finally the interpolated image using the smoothed flow

interpolated images using the uncorrected flow and the smoothed flow get RMS values that are usually lower, which shows that usually the image quality of the smoothed flow version is higher than the uncorrected one. The last result in the table gives us results for the images that are close to each other but visually, the artifacts in the smoothed version are much less noticeable than the ones we have in the uncorrected version. Such a situation can be seen in 7.1, where even though the smoothed version (bottom image) also has artifacts, the overall image feels much cleaner and realistic than the uncorrected version (top image). Finally the gain of the images obtained with the smoothed field becomes much more obvious in dynamic cases than on static images, which is what we aimed for.

In Figure 7.4 we show a transition sequence of our results on an exterior sequence. The top and bottom images are the origin and destination images, and the other images are transition frames, with coefficients w = c = 0.2, 0.4, 0.5, 0.6 and 0.8 from the top to bottom image respectively. We see from this figure that even though there are a few artifacts, the quality of the interpolated frames is very good and the geometry of the scene is well preserved during the interpolation.

Figure 7.1: This image shows an example of the visual improvements that are attained by using the smoothed optical flow (bottom image) compared to the uncorrected version (top image).

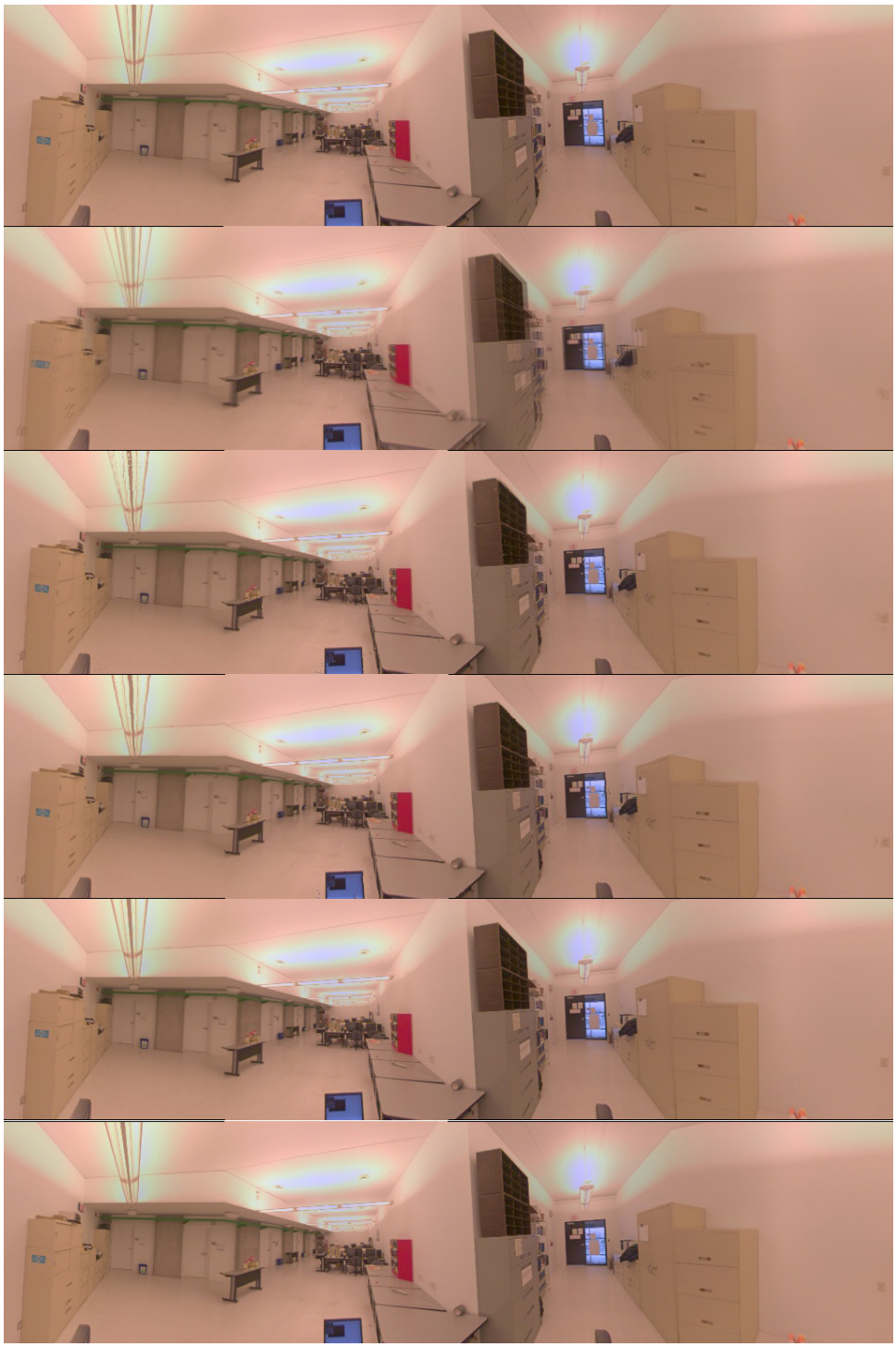

Figure 7.2: Going from the top to bottom image, we have: the origin image; the linearly interpolated image; the uncorrected flow interpolated image; the interpolated image using a smoothed flow; the real image captured at the interpolated location (ground truth); and the destination image. All interpolation uses the best matching parameter $c = 0.45$.

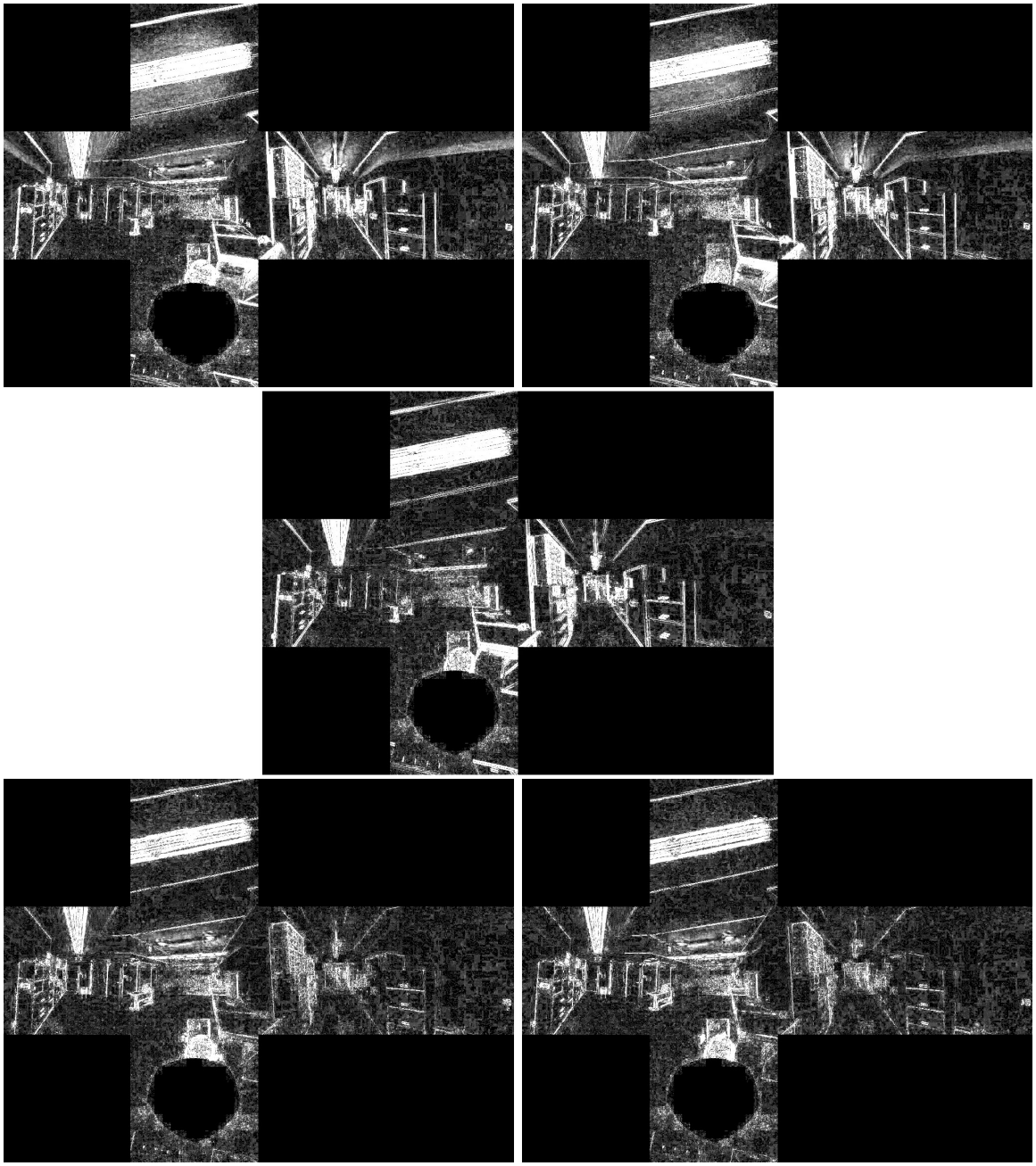

Figure 7.3: Going from top to bottom image (and left to right), we have the comparison images of the real panorama captured at interpolation location with: the origin image; the target image; the linearly interpolated image; the interpolated image using uncorrected flow; and finally with the interpolated image using the smoothed flow. All interpolated images use the best matching parameter $c = 0.45$. This is the same sequence as the one in Figure 7.2

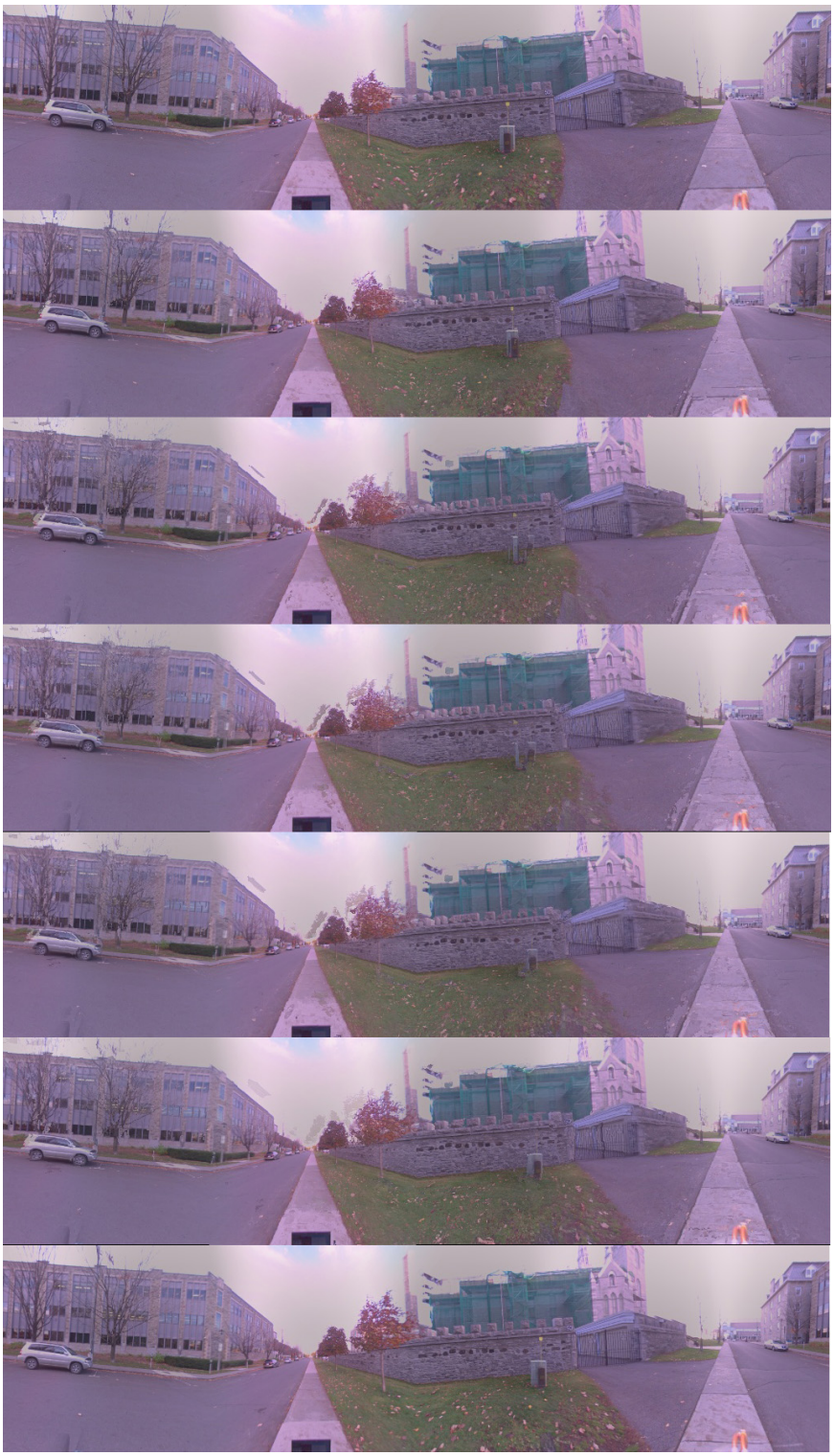

Figure 7.4: An outside sequence. Top and bottom images are the captured origin and destination. The other images are (from top to bottom) interpolated images using the smoothed optical flow field with c=0.2, 0.4, 0.5, 0.6 and 0.8.

7.3 On The Use of Epipolar Constraint

In the previous chapter we showed that only reprojecting the calculated optical flow on the epipolar lines was not sufficient and could even decrease the quality of the results. However our second approach was more successful: we constrained the search of the optical flow to those candidate shift that were close to the estimated epipolar lines. This didn't increase the overall quality of our images, but on certain parts of the images such as the one shown in 7.5 the improvements are very noticeable and encouraging. This figure shows on the left the transition images of a certain pair of images, interpolated using a flow with epipolar constraining and smoothing, and on the right the interpolated sequence using a flow with only the smoothing correcting. We can see that even though there are artifacts in our epipolar constraining, the geometry of the scene is preserved much better. Even though this is not usable yet, it shows great promise for future work and should be an area to focus on to improve the results of our interpolation.

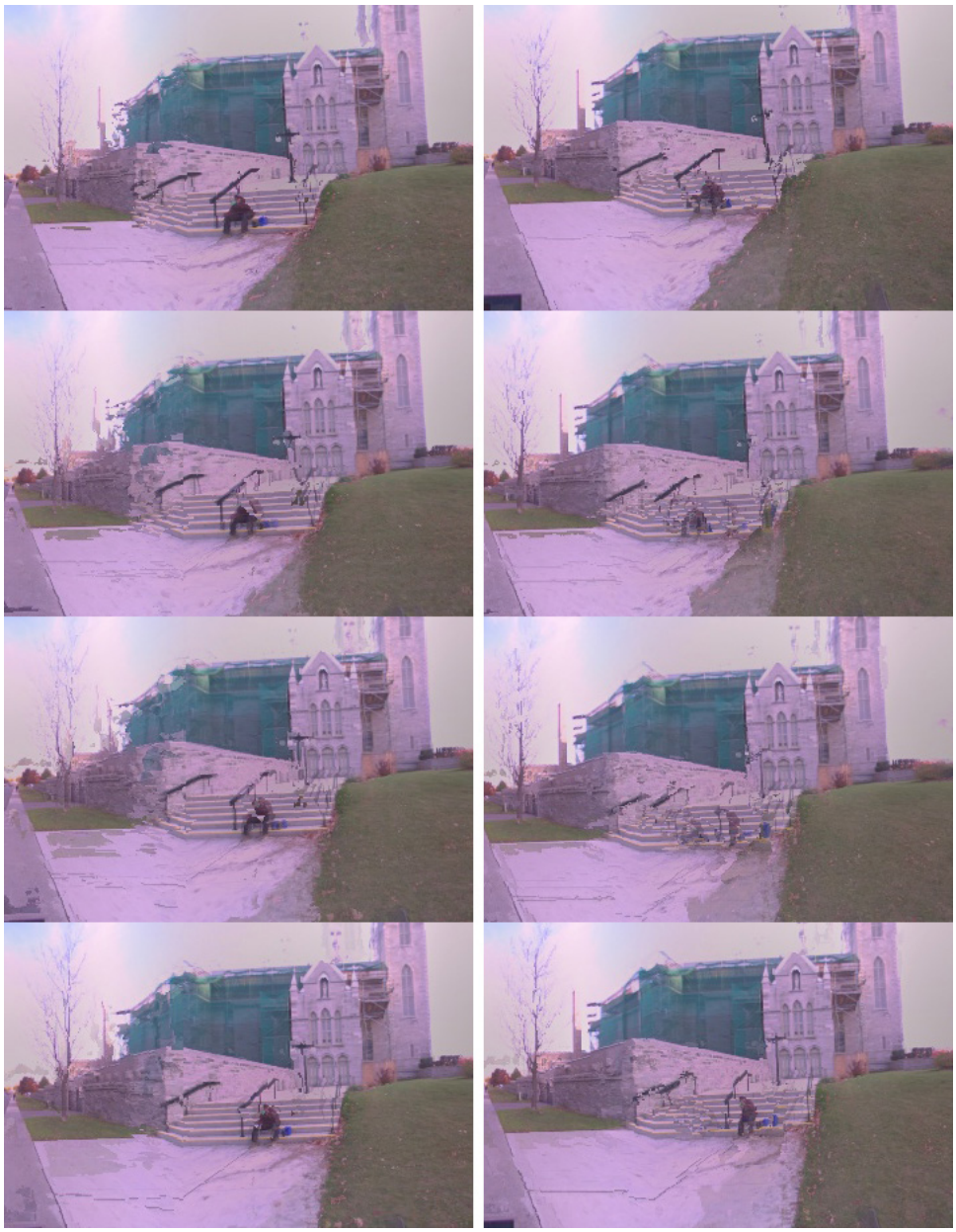

Figure 7.5: A Zoom on a part of our transition sequence. On the left the transition images of a certain pair of images, interpolated using a flow with epipolar constraining and smoothing, and on the right the interpolated sequence using a flow with only the smoothing correcting.

7.4 On The Importance of Small Distances between panoramas

All along this thesis we have used the assumption that our panoramas have to be captured at short distances from one another to ensure that the best interpolation quality possible is attained. To illustrate this, we have ran another set of experiments. We use 4 panoramas: I_0, I_1, I_2 and I_3 from a sequence of panoramas where all the panoramas were captured close to each other. From these panoramas, we interpolate 3 transition images, to illustrate the degradation of the interpolation quality with the increase of the distance between viewpoints. These images will be I_{t_0}, I_{t_1} and I_{t_2} which will be interpolated from I_1 and I_2, I_0 and I_2 and finally I_0, I_3 respectively. To ensure that the comparison is possible, we tried that all the interpolated images represented the same physical position, to achieve this, we used the following weights w = c = 0.5, 0.75 and 0.5 for I_{t_0}, I_{t_1} and I_{t_2} respectively. Figure 7.6 illustrates our test set: we represent each sequence with a different step value (1, to and 3 from top to bottom), where the panoramas drawn in full represent the panoramas used for the interpolation and the dashed are the other panoramas of the sequence but are not used. The red crosses represents the estimated position of the interpolated panorama. It is also important to note that the increase of the distance between viewpoints doesn't only have an impact on the quality of the interpolation, it also decreases the speed at which we can calculate the flow because the maximum displacements intervals in the texture are bigger. In our experiments, for both x and y we used an interval of [-40, 40] for the flow field calculation of I_{t_0}, [-80, 80] for I_{t_1} and finally [-120, 120] for I_{t_2}. We didn't apply any smoothing to the flow field to further illustrate the degradation of the results. In Figure 7.7 we show the interpolation image that we retrived from an indoors sequence were the panormas were captured very close to each other. The degradation in this image is noticeable but not major. In Figure 7.8 we ran the same test on different sequence of outdoors panoramas, that were taken further away from each other. In this case the degradation of the image if much more noticeable.

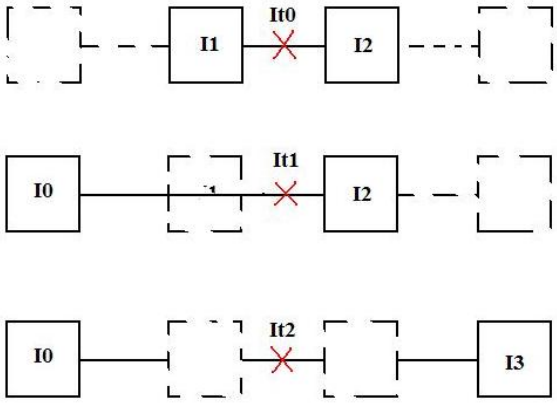

Figure 7.6: This image shows the position of the different interpolated images that we used to demonstrate the degradation of the interpolation quality with the increase of the distance between viewpoints.

Figure 7.7: This image illustrates the degradation in the flow calculation algorithm with the increase of the distance between viewpoints from an indoors capture sequence.

Figure 7.8: This image illustrates the degradation in the flow calculation algorithm with the increase of the distance between viewpoints from an outdoors capture sequence.

7.5 On the importance of Static Scenes

The second assumption that we use in our method is the fact that the scene has to be static. In this section we will give an example that illustrates the importance of this assumption. In this example, we have a car that drives by our viewpoint while we are capturing the environment. Figure 7.9 shows the interpolation resulting from this image, using uncorrected flow. From top to bottom, we show the origin image, the interpolated images with coefficient w = c = 0.2, 0.5 and 0.7 respectively and the destination image. We can see that the car disappears and suddenly reappears in the scene, and this is due to the fact that the displacement of the object is too big and was not captured by our optical flow interval. If the movement of the object had been smaller such as the one seen in Figure 7.10 the movement would have been captured and the object wouldn't have had such artifacts. This problem is not present only with moving objects. It is actually related to parts of the scene where the displacement is too big to be captured by the interval chosen in the optical flow algorithm: such a case can be seen in Figure 7.9, where the grass close to the viewpoints suffers from the same kind of ghosting artifacts as the moving car. One possible solution would be increasing the interval used in the optical flow algorithm, but this wouldn't work for the car in this case because the car is changing face, and the extended cube representation used does not capture enough of the scene to see the destination of every of the pixels of the car. Also, even if we could increase the size of the cube, we are limited to capturing at most half of the adjacent face, at the cost of huge images and processing times, and this would still not be enough to be able to follow any moving object at any moving speed. Thus a more general solution needs to be found to deal with these cases. Such a solution would also allow us to dramatically increase the distance between two consecutive viewpoints.

7.6 Conclusion

In this Chapter, we presented some of the results of our method. In the next Chapter we will conclude on our work and we will present a few possible orientation for future research.

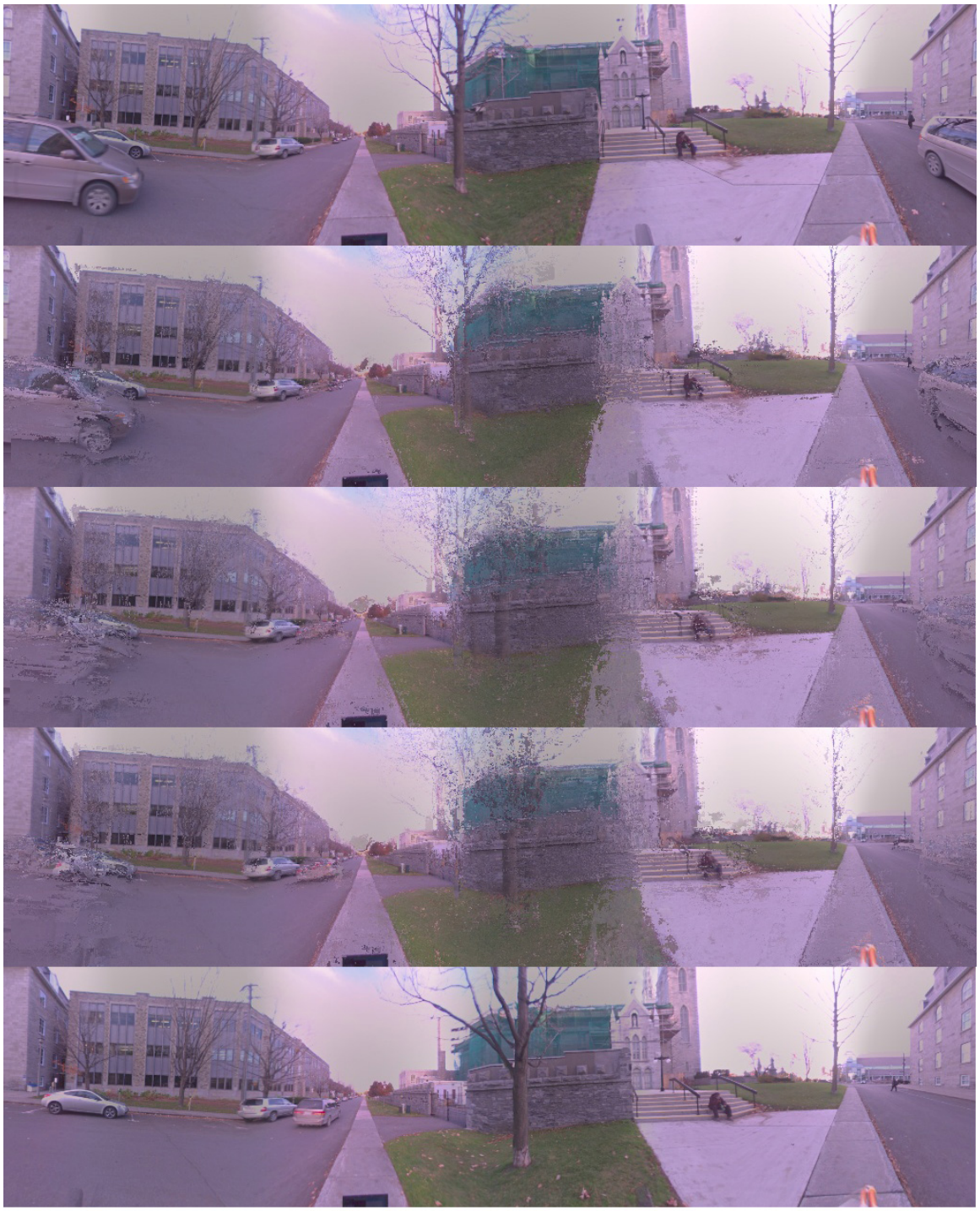

Figure 7.9: This image illustrates the artefacts that occur when a moving object is present in our scene, such as the car in this example.

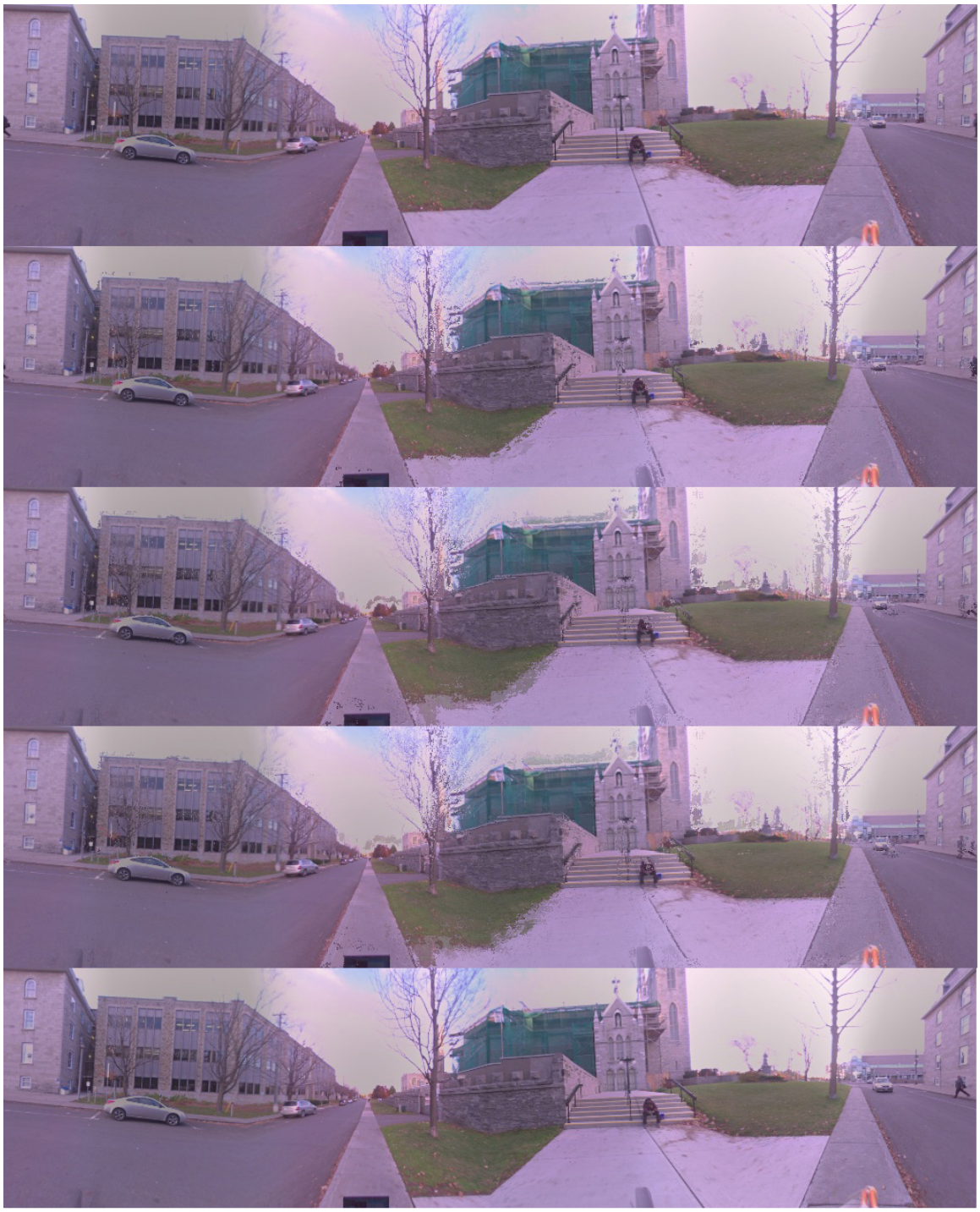

Figure 7.10: This image illustrates the artefacts that occur when a moving object is present in our scene, such as the car in this example.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we have presented a new way of interpolating between pairs of panoramas in real-time using the GPU, which allows us to navigate inside a scene and achieve a high degree of realism. Our main contribution concerns the interpolation of intermediate viewpoints of a scene in real-time using a computed optical flow field. Except for a few artifacts, our results are of good quality, as long as the panoramas were taken at reasonable distances from each other.

We first presented an algorithm to calculate the optical flow on planar surfaces, and then we experimented with various methods designed to enhance the dense optical flow field between pairs of cubic panoramas. Some of these methods show undeniable improvements while other were less successful, but overall the quality of our optical flow was improved. Our next improvements consisted in allowing real-time navigations between a sequence of panoramas. This was successfully achieved by synthesizing new panoramas on the GPU at a rate of at least 300 panoramas per second on slow GPUs (at least 5 years old). But the unique implementation on the GPU was not enough to achieve real-time navigation, and it wouldn't have been possible to navigate through a series of possibly an infinity of viewpoints without these 3 additional steps: buffering of the data, multi-threading of the application and preprocessing of the optical flow and its corrections.

Our interpolation algorithm can synthesize any number of new viewpoints given a pair of panoramas, and the different panoramas can have any resolution desired by the user, the only restriction being that a dense optical flow be made available for each of the pairs

of panoramas. The transition sequences we synthesize are of high quality, but still present some artifacts. By keeping in mind that the user cannot pause between panoramas, most of the artifacts that occur during the synthesis of individual transition panoramas are not noticeable during the navigation, but others are and these need to be corrected. It is the authors' belief that this problem would be completely corrected if a completely accurate optical flow algorithm was devised. This is one of the advantages of our method: the quality of the navigation will increase with the quality of more recent optical flow without any additional effort being necessary to change our navigation software.

8.2 Future Work

There are many improvements that can be added to our approach to enhance the immersion of the user and the quality of the immersion in any image-based virtual navigation software. Some possible future work include but are not limited to the following:

1. Improve the quality of the flow field and make it faster to compute. Improving the quality of the flow field would remove most/all of the artifacts present in our generated transition images, which would in turn greatly increase the immersion of the users.

2. Achieve real-time navigation not only on a selected path, but in the whole space where the views have been taken. If this was achieved, the sense of immersion of the user would be improved and this one would be able to freely move inside the scene.

3. Since this architecture was designed to run on fairly old GPU, it would be very interesting and fairly straightforward to develop a browser based framework to give access to our navigation software to a wide audience.

4. Reduce the amount of data that is required at the moment. This is not such a problem at the moment, but with the extension to online use, we would need to transfer as little data as possible to be able to keep a high frame rate.

5. Find a way to acquire the data where the black hole is present in each of our panoramas. This black hole is caused by the fact that we only use a single Ladybug camera that is mounted on another device to move it around, and this device is in the area where the black hole appears. First this creates a setback in the immersion

side of our software and most importantly, in certain cases, can make our capture loose important parts of the scene, as it would be the case if we were capturing the Marble Court in the Palace of Versailles in France, to only name one example.

Appendix A

Environment Representation

As we explained before, we capture our environment through a series of panoramic images. These images are taken in sequence on a path followed by the camera. Thus we can represent our environment as a 2 way graph, where the nodes are the different viewpoints captured and the edges are the paths that can be taken by the user of our viewer (See Figure A.1 for an example).

Within the architecture of our method, we are not limited to any number of paths or any number of panoramas. Also some panoramas can be captured and added later to our virtual environment, and their connection to the already existing panoramas can be added easily.

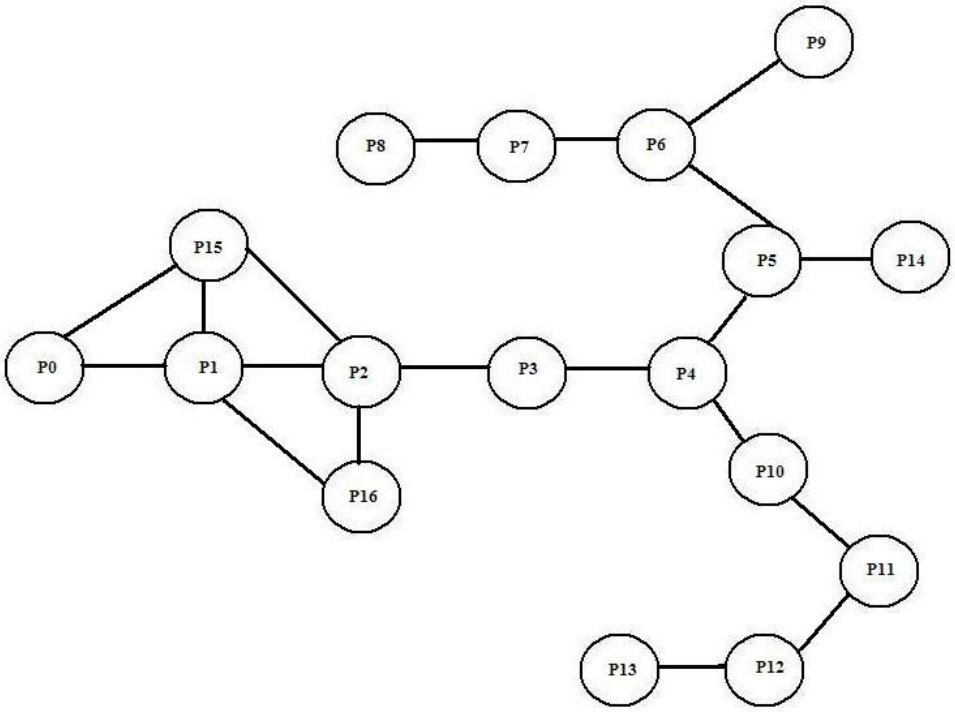

Figure A.1: An example graph representing the environment captured during one of our capture sessions.

Appendix B

The NAVIRE Viewer

To demonstrate our real-time navigation method, we have developped the NAVIRE viewer, which allows us to navigate in any environment that was previously captured. In this appendix, we will describe the viewer's user interface and the different features that were made available to enhance the experience of the user.

The first important feature that is present in our viewer is the navigation window (Figure B.1), which allows us to see a cubic panorama, and look around our view to different parts of the scene. It also allows us to navigate between different cubes. The next feature is the option to enable/disable 'smooth' navigation, which allows us to advance through multiple cubes in sequence. The smooth navigation will stop when we come to an intersection. At this point a decision needs to be made by the user as to which direction to take next. To increase the user ease to navigate in the environment, a map of the environement is provided which rotates depending on the direction in which the user is looking. The position of every cube in our sequence is also represented on the map, every cube is drawn in white except for the cube that the user is currently viewing, which is drawn in a different color (red in our case). Since the map is small, it is sometimes difficult to identify which cube we are currently looking at on the map. To alleviate this problem we provided 2 enhancements: the user can zoom in and out of the map with the focus around the current panorama; the user can display the map in full screen on top of the navigation window (Figure B.2).

Another feature that is present in our viewer architecture is an interface to position on the map the sequence of cube that we are viewing (Figure B.3). This allows the user to select a interval of cubes, and position them onto the map in a straight line (if the path taken is square, we need to do this 4 times) and save it. Once the map has been

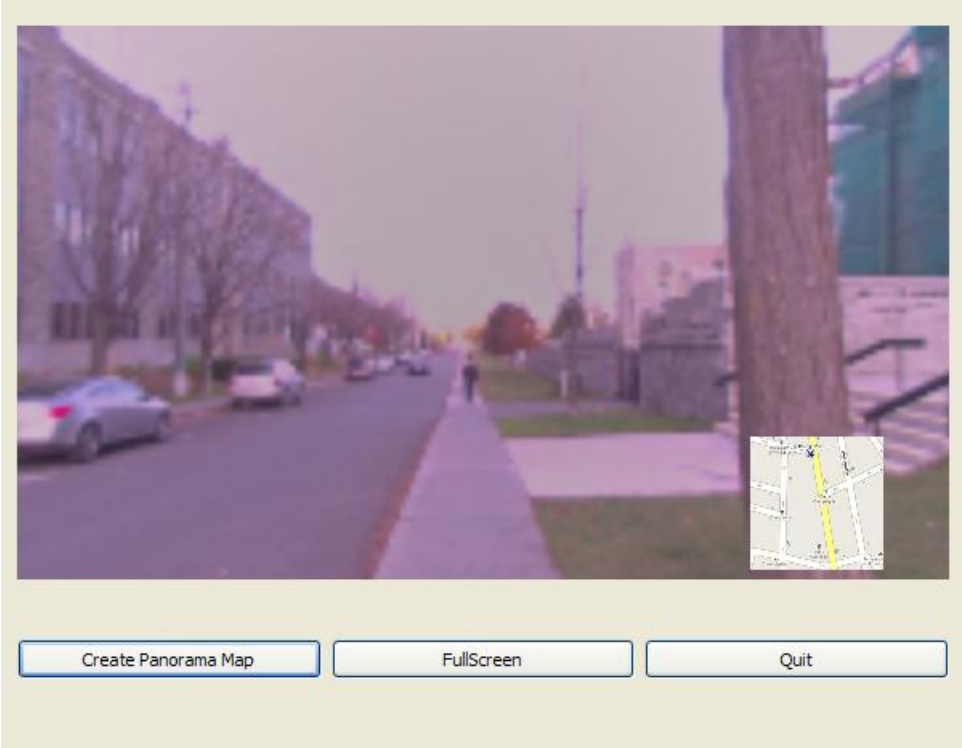

Figure B.1: In this figure we can see the navigation window and GUI of the NAVIRE viewer.

created and we return to the navigation window, the map is automatically updated, and the new position in the list of cubes is visible and used.

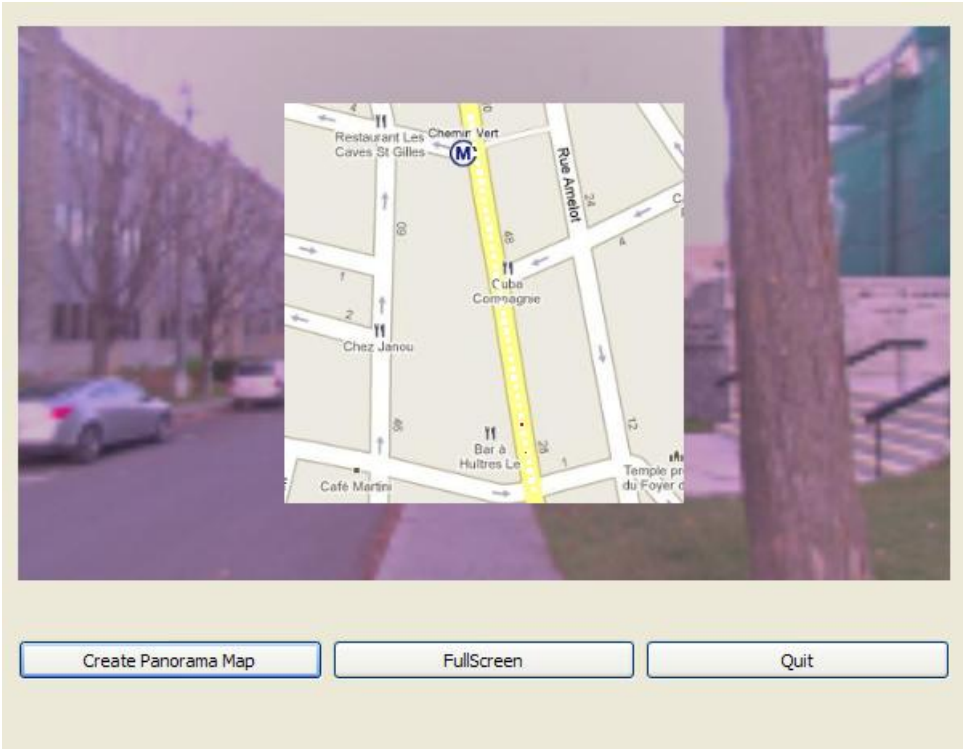

Figure B.2: In this figure we can see the one of our map feature, which allows us to see the map full screen for a more detailed view.

Figure B.3: This figure shows the map creation interface of our application.

Bibliography

[1] Nvidia cuda, programming guide.

[2] Ptgui. http://www.ptgui.com/gallery/, 2010.

[3] Mario Sormann Andreas Klaus and Konrad Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *Proceedings of the 18th International Conference on Pattern Recognition*, 2006.

[4] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer Vision and Image Understanding (CVIU)*, 110:346–359, 2008.

[5] S. Birchfield and C. Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20(4):401–406, 1998.

[6] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. Description of the algorithm, 2002.

[7] Shenchang Eric Chen. Quicktime vr: an image-based approach to virtual environment navigation, 1995.

[8] Eric Dubois Feng Shi, Robert Laganiere and Frederic Labrosse. On the use of ray-tracing for viewpoint interpolation in panoramic imagery. In *Proceedings of the 2009 Canadian Conference on Computer and Robot Vision*, pages 200–207, 2009.

[9] Y. Furukawa and J. Ponce. Pmvs. http://www.cs.washington.edu/homes/furukawa/research/pmvs

[10] R. Gupta and S.-Y. Cho. Real-time stereo matching using adaptive binary window. In *3DPVT*, 2010.

[11] R. Hartley. In defence of the 8-point algorithm. In *Proceedings of the Fifth International Conference on Computer Vision*, pages 1064–1070, 1995.

[12] Richard I. Hartley. Theory and practice of projective rectification. *International Journal of Computer Vision*, 35:115–127, 1999.

[13] Richard Hartley Hongdong Li. Five-point motion estimation made easy. *Proceedings of the 18th International Conference on Pattern Recognition*, 1:630–633, 2006.

[14] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[15] T.S. Huang and A.N. Netravali. Motion and structure from feature correspondences: A review. *Proceedings of the IEEE*, 82:252–268, 1994.

[16] J. Illingworth and J. Kittler. A survey of the hough transform. *Computer Vision, Graphics, and Image Processing*, 44:87–116, 1988.

[17] Paul F. Whelan John Mallon. Projective rectification from the fundamental matrix. *Image and Vision Computing*, 23:643–650, 2005.

[18] G. Lafruit R. Lauwereins K. Zhang, J. Lu and L. Van Gool. Real-time accurate stereo with bitwise fast voting on cuda. In *ICCVW*, 2009.

[19] Sing Bing Kang and Heung-Yeung Shum. A review of image-based rendering techniques. In *Visual Communications and Image Processing*. Institute of Electrical and Electronics Engineers, Inc., 2000.

[20] Florian Kangni and Robert Laganiere. Rectification and pose recovery for spherical images. Master's thesis, University of Ottawa (SITE), 2007.

[21] S. Kolhatkar and R. Laganiere. Real-time virtual viewpoint generation on the gpu for scene navigation. *CGI 2010*.

[22] S. Kolhatkar and R. Laganiere. Real-time virtual viewpoint generation on the gpu for scene navigation. *Computer and Robot Vision (CRV), 2010 Canadian Conference on*, pages 55 – 62, May 31 2010-June 2 2010.

[23] Maxime Lhuillier and Long Quan. Image interpolation by joint view triangulation. In *IN PROCEEDINGS OF THE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, FORT COLLINS*, pages 139–145, 1999.

[24] Hongdong Li. A simple solution to the six-point two-view focallength problem. In *In European Conference on Computer Vision*, pages 200–213, 2006.

[25] John Lim and Nick Barnes. Directions of egomotion from antipodal points. In *Computer Vision and Pattern Recognition (CVPR)*, 2008.

[26] John Lim and Nick Barnes. Estimation of the epipole using optical flow at antipodal points. *Computer Vision and Image Understanding (CVIU)*, 114:245–253, 2010.

[27] Charles Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision, 1999.

[28] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, pages 674–679, 1981.

[29] David Luebke and Greg Humphreys. How gpus work, 2007.

[30] Jing Wang Xiaoxun Zhang Luping An, Yunde Jia and Mingxiang Li. An efficient rectification method for trinocular stereovision. *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04)*, 4:56–59, 2004.

[31] Robert C. Bolles Martin A. Fischler. *Readings in computer vision: issues, problems, principles, and paradigms*, chapter Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography, pages 726–740. Morgan Kaufmann Publishers Inc., 1981.

[32] Wojciech Matusik, Matthias Zwicker, and Frdo Durant. Texture design using a simplicial complex of morphable textures. In *SIGGRAPH*, 2005.

[33] Tomaso Poggio Michael J. Jones. Multidimensional morphable models: A framework for representing and matching object classes. *International Journal of Computer Vision*, 29:107–131, 1998.

[34] David Nistr. An efficient solution to the five-point relative pose problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:756–777, 2004.

[35] A. S. Ogale and Y. Aloimonos. Shape and the stereo correspondence problem. *International Journal of Computer Vision*, 65:147 – 162, 2005.

[36] Abhijit S. Ogale and Yiannis Aloimonos. A roadmap to the integration of early visual modules. *International Journal of Computer Vision*, 72:9 – 25, 2007.

[37] Ruigang Yang Henrik Stewnius David Nistr Qingxiong Yang, Liang Wang. Stereo matching with color-weighted correlation, hierarchical belief propagation, and occlusion handling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:492–504, 2009.

[38] Jan-Michael Frahm Rahul Raguram and Marc Pollefeys. A comparative analysis of ransac techniques leading to adaptive real-time random sample consensus. *Proceedings of the 10th European Conference on Computer Vision: Part II*, 5303:500–513, 2008.

[39] Mark A. Ruzon and Carlo Tomasi. Edge, junction, and corner detection using color distributions. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 23:1281–1295, 2001.

[40] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence. *International Journal of Computer Vision*, 47:7 – 42, 2002.

[41] Steven M. Seitz. Toward interactive scene walkthroughs from images. In *Proceedings of the 1998 Workshop on Computer Vision for Virtual Reality Based Human Communications (CVVRHC '98)*, 1998.

[42] Steven M. Seitz and Charles R. Dyer. View morphing. In *SIGGRAPH*, 1996.

[43] Hans-Peter Seidel Sergey Kosov, Thorsten Thormhlen. Accurate real-time disparity estimation with variational methods. *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, 5875:796–807, 2009.

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

[45] Feng Shi. Panorama interpolation for image-based navigation. Master's thesis, University of Ottawa (SITE), 2007.

[46] J. Shi and C. Tomasi. Good features to track. Technical report, Cornell University, 1993.

[47] N. Snavely. Bundler: Sfm for unordered image collections. http://phototour.cs.washington.edu/bundler/.

[48] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. *ACM Transactions on Graphics (SIGGRAPH Proceedings)*, 25:835–846, 2006.

[49] Changming Sun. Uncalibrated three-view image rectification. *Image and Vision Computing*, 21:259–269, 2003.

[50] Xiaoyong Sun and Eric Dubois. View morphing and interpolation through triangulation. *Image and video communications and processing*, 5685:513–521, 2005.

[51] Inigo Thomas, Inigo Thomas, Eero Simoncelli, and Eero Simoncelli. Linear structure from motion. Technical report, University of Pennsylvania, 1994.

[52] Tina Y. Tian, Carlo Tomasi, and David J. Heeger. Comparison of approaches to egomotion computation. In *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition*, pages 315–320, 1996.

[53] Z. Wang and Z. Zheng. A region based stereo matching algorithm using cooperative optimization. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2008.

[54] L. Xu and J.Y. Jia. Stereo matching: An outlier confidence approach. In *ECCV08*, pages IV: 775–787, 2008.

[55] S. M. Seitz Y. Furukawa, B. Curless and R. Szeliski. Manhattan-world stereo. CVPR, 2009.

[56] Steven M. Seitz Yasutaka Furukawa, Brian Curless and Richard Szeliski. Reconstructing building interiors from images. In *Twelfth IEEE International Conference on Computer Vision (ICCV 2009)*, 2009.

[57] Huaifeng Zhang, Jan Cech, Fuchao Wu, and Zhanyi Hu. A linear trinocular rectification method for accurate stereoscopic matching. In *in British Machine Vision Conf*, pages 281–290, 2003.

[58] Heung-Yeung Shum Ziqiang Liu, Ce Liu and Yizhou Yu. Pattern-based texture metamorphosis. In *PG'02: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, pages 184–191, 2002.