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

Shanat Kolhatkar, Robert Laganière
School of Information Technology and Engineering (SITE),
University of Ottawa
800 King Edward, P.O. Box 450, Stn A
Ottawa, Ontario, Canada, K1N 6N5
shanat.kolhatkar@gmail.com, laganier@site.uottawa.ca

Abstract

In this paper 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.

1. Introduction

There has been a lot of interest from the computer graphics and the computer vision fields towards image-based modeling and rendering methods [2]. 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. [14] 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 paper 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.

Section 2 presents the previous work relevant to our method. Section 3 introduces the optical flow calculation between a pair of images and Section 4 discusses the extension to cubic panoramas. Section 5 describes the view interpolation algorithm. Section 6 provides some implementation details required to achieve real-time calculation of the transitions. Section 7 presents some results and Section 8 is a conclusion.

2. Related Work

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 [5] and [7]. 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.

One of the base paper in the field of view interpolation is [10]. 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 [4], [13]. Both approaches triangulate the images using a certain set of points in the images, which can be chosen manually or automatically using for example [12]. Once the triangulation is done, the triangles of the images are warped and the colors are interpolated to get a transition image. Another paper related to the topic of view interpolation is [11], which uses a ray-tracing approach to synthesize new views from an input pair of cubic panoramas. A downside of this approach is that it is time consuming and the resulting interpolated images present many artifacts.

A good example of 3D reconstruction can be seen in [9]. 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.

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

2.2. Optical Flow

Many approaches to evaluate the optical flow between pairs of images have been proposed. Some classic methods are [3], [6], [1]. 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 [8]. 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.

3. Optical Flow Estimation

As we mentioned before, our virtual view point generation system uses the optical flow field between reference images as input to the interpolation process. In this section we will briefly 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 [8] 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 fonction 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. To achieve a contrast-invariant matching, the goodness function locally compares pixels according to the phase difference between the two images: the closest the phase difference is to 0, the more similar the pixels are. The phase difference information is obtained by using multiple Gabor filters of different scales, spatial frequencies and orientations on both images, which are then combined and averaged to get the final comparison value. The measure is used for local matching because, according to the authors of the methods, it helps detect depth discontinuities more accurately. For more detailed information on the goodness function evaluation and the contrast-invariant matching measure, see [8].

We selected this algorithm because it generally gives good results. 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.

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

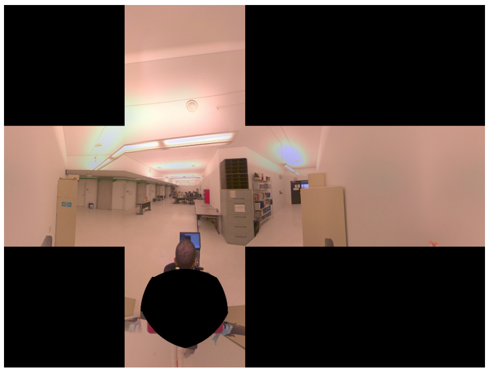

Figure 1. Example of a cubic panoramas, displayed in an unfolded form

4. Estimating the optical flow on panoramic images

Our approach would work on standard imagery, but to enhance the sense of immersion of the user and the realism of the scene rendering, we have chosen to use 360° panoramas. Different representations are available to manipulate the panoramas: cubic, spherical or cylindrical panoramas. In cubic panoramas (see Figure 1), 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 these panoramas. Also, 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.

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 es-

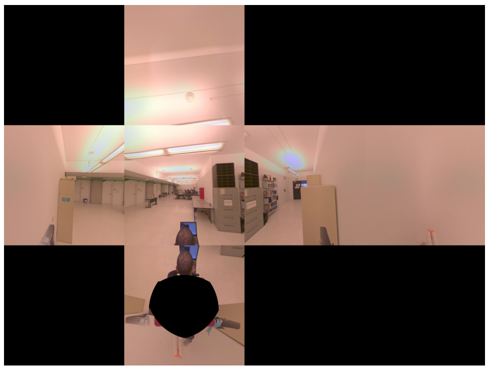

Figure 2. Example of our extended cubic panoramas, displayed in an unfolded form.

timate the displacement of a visual point moving from one face to another. One such extended cube is shown in Figure 2 where it can be seen that some elements on one face are repeated on adjacent faces. 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. Note that, when proceeding this way, it happens that some displacement vectors are computed twice (Figure 4), because they appear on each extended portion of two adjacent faces (Figure 3). 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.

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 $i \leq j$, $k \leq l$, $i, k \leq 0$, $j, l \geq 0$. We summarized this step with the following pseudocode, using the same no-

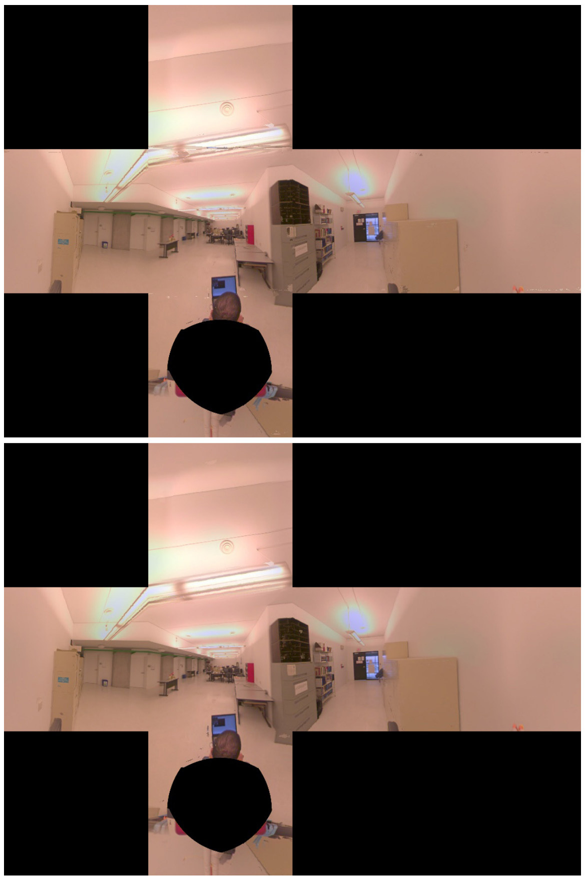

Figure 3. This figure shows an interpolated frame computed from a normal cubic panorama (top) and from an extended cubic panorama (bottom) at the same view position. Both are unfolded view of 256x256 face images. Some mistakes can be seen in the interpolation from normal cubes that are not present in the extended cubic representation (e.g. the fluorescent on the ceilling).

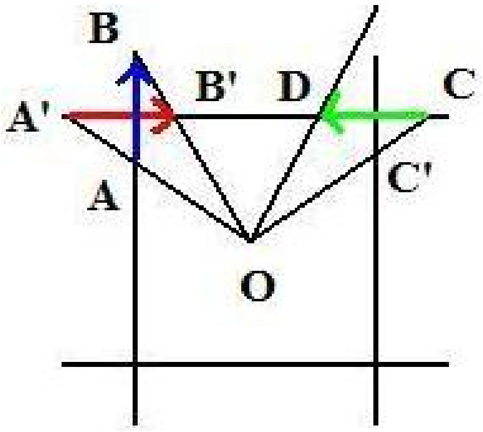

Figure 4. This figure illustrates the geometry of the extended cubic representation drawn in 2D. A' is the projection of A on a different faces, and the red and blue vectors are their respective flow vectors on each face. In this case, only the best displacement vector is kept. The green vector is one that does not have a corresponding displacement vector on the other face and does not require additional processing.

tations as previously:

```
Get N, the nxn color neighborhood of o in I1
forall 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 $I_2$, p' =
            $(o_x + d_x + offset_x, o_y + d_y + offset_y)$
            Get N', nxn color neighborhood of p' in $I_2$
            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. These sparse outliers in the displacement field can cause very annoying effects. This is illustrated in Figure 5.

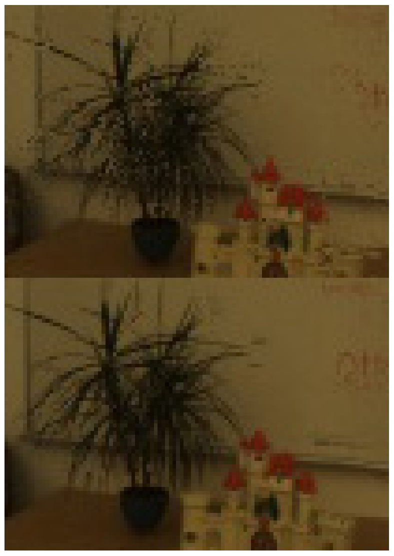

Figure 5. The images are selected parts of a two interpolated images with interpolation coefficient 0.5. On the top interpolated without smoothing and on the bottom we applied the smoothing step. We can see that without smoothing the scene looks blurry and has many artefacts, whereas it keeps its sharpness and shape when using the smoothing step.

5. View Interpolation

Our view interpolation is based on the algorithm developed in [7]. This algorithm was designed to work on single 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.

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 previously. 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 creates a lot of artifacts, and the quality of the results can be unpredictable and unrealistic, especially because this approach doesn't take into account the geometry of 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 interpolate the displacement vectors. The morphing between pairs of images uses the following equation:

$$I_t(x,y) = (1-c)I_0(x_0,y_0) + cI_1(x_1,y_1) \qquad (1)$$

where $I_t(x,y)$ is the pixel of coordinate (x,y) in the transition image, $I_0(x_0,y_0)$ and $I_1(x_1,y_1)$ are the pixels of image I_0 and I_1 respectively. We also define $(x_0,y_0) = (x+d_x, y+d_y)$ and $(x_0,y_0) = (x+d'_x, y+d'_y)$: the displaced coordinates for pixel (x,y) in I_0 and I_1 respectively. We set $d = (dx,dy) = (wW_{01}(x,y))$, $d' = (d'x, d'y) = ((1-w)W_{10}(x,y))$, with $W_{01}(x,y)$ and $W_{10}(x,y)$ being the displacement vector from images I_0 to I_1 and I_1 to I_0 respectively for pixel (x, y). $W_{01}(x,y)$ and $W_{10}(x,y)$ are obtained from the optical flow calculation with $W_{01}(x,y) = -W_{10}(x,y)$. Finally 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 in a similar fashion in order to give the impression of movement between the views.

6. Real-Time Implementation

Real-time navigation 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. 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. We therefore need to define an efficient strategy to retrieve that data and thus minimizing 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 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.

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 ran on the GPU, to calculate the interpolated images. This scheme allows us to constantly have a thread in the background checking for the necessary data and loading it, without the user noticing any downtime.

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);
$offset_y$ *= fDisplacementIntervalLengthY;
vec2 $texcoordOrigin =$
$(iPixCoord - fDispIC * offset)/fTexSz$;
vec2 $texcoordTarget =$
$(iPixCoord+(1.0f-fDispIC)*offset)/fTexSz$;
//Interpolate the colors
$gl_FragColor = (1.0f - fColorIC) *$
$texture2D(OriTex, texcoordOrigin) +$
$fColorIC * texture2D(TarTex, texcoordTarget)$;
```

7. Results

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.

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 few 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 6. 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 6, 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, we compare the interpolated extended cube panoramas with the ground truth image from the same sequence as Figure 6. We compare the images pixel by pixel and set to black all pixels that are similar to the reference panorama. For each of the images, we get a root mean square of 22.61, 21.49, 17.55, 14.02, 14.05 when comparing 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 show that our interpolated images are much similar to the actual image that the linearly interpolated image. The interpolated images using the uncorrected flow and the smoothed flow are very close to each other, even though the smoothed version might give images with slightly more artefacts, visually, they are much less noticeable than the ones we have in the uncorrected version. And 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.

8. Conclusion and Future Work

In this paper, 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 concern the interpolation of intermediate viewpoints of a scene in real-time using 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.

Some possible future works would be to improve the quality of the flow field and make it faster to compute. Another enhancement of great interest would be the ability to achieve real-time navigation not only on a selected path, but in the whole space where the view 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.

References

[1] J.-Y. Bouguet. Pyramidal implementation of the lucas kanade feature tracker. 2002.

[2] S. K. H-Y Shum. A review of image-based rendering techniques. 4067:2–13, June 2000.

[3] B. K. Horn and B. G. Schunck. Determining optical flow. artificial intelligence. 17:185–203, 1981.

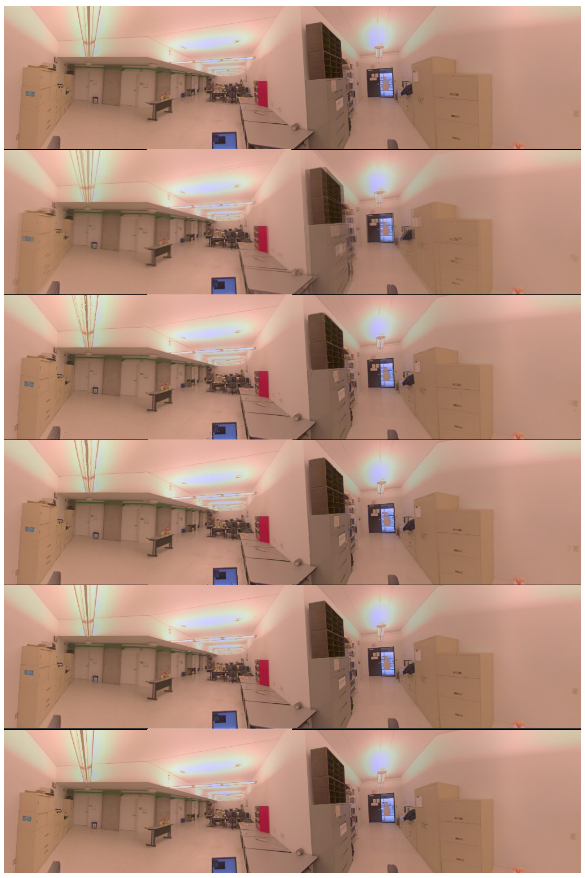

Figure 6. Going from the top to bottom image, we have: the origin image; the linearly interpolated image; the uncorrected flow interpolated image; the interpolated image using a smoothed flow; the real image captured at the interpolated location (ground truth); and the destination image. All interpolation uses the best matching parameter $c = 0.45$.

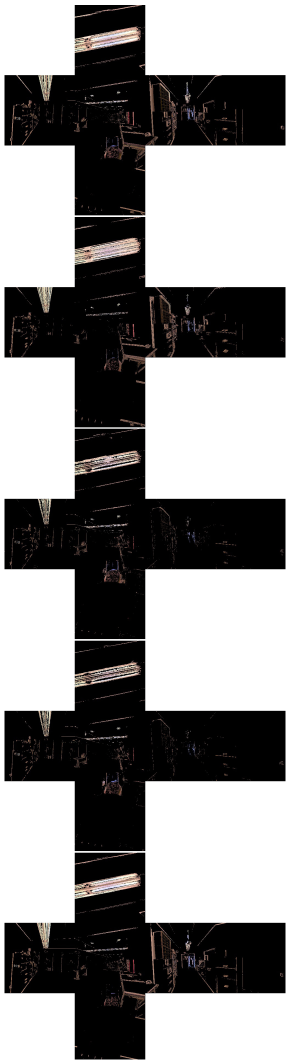

Figure 7. Going from top to bottom image, we have the comparison images of the real panorama captured at interpolation location with: the origin image; the linearly interpolated image; the interpolated image using uncorrected flow; the interpolated image using the smoothed flow; and finally with the target image. All interpolated images use the best matching parameter $c = 0.45$. This is the same sequence as the one in Figure 6

[4] M. Lhuilliier and L. Quan. Image interpolation by joint view triangulation. 2, 1999.

[5] Z. Liu, C. Liu, H.-Y. Shum, and Y. Yu. Pattern-based texture metamorphosis. 0:184–191, 2002.

[6] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. 17:674–679, 1981.

[7] W. Matusik, M. Zwicker, and F. Durant. Texture design using a simplicial complex of morphable textures. 4:124–125, 2005.

[8] A. S. Ogale and Y. Aloimonos. A roadmap to the integration of early visual modules. 72:9–25, April 2007.

[9] S. M. Seitz. Toward interactive scene walkthroughs from images. 1998.

[10] S. M. Seitz and C. R. Dyer. View morphing. 1996.

[11] F. Shi, R. Laganiere, E. Dubois, and F. Labrosse. On the use of ray-tracing for viewpoint interpolation in panoramic imagery. 2009.

[12] J. Shi and C. Tomasi. Good features to track. 1994.

[13] X. Sun and E. Dubois. View morphing and interpolation through triangulation. 2005.

[14] S. M. S. Yasutaka Furukawa, Brian Curless and R. Szeliski. Reconstructing building interiors from images. 2009.