Computing optical flow using fast total variation

Arthur Masson

Computing optical flow using fast total variation

Author: Arthur Masson
Supervisor: Ingo Tzschichholtz
Company: VITRONIC

University of Rennes 1

August 13, 2011

Contents

1 Acknowledgments

First of all, I would like to thanks Ingo Tzschichholtz, my project supervisor who supported my work and gave me the opportunity to dive into the world of both research and applied computer vision. M. Ingo gave me enough freedom to do some important strategic choices, which made my work creative and instructive.

The internship would not have been possible without Ewa Kijak who I thank gratefully. I also thank Stefan Gering, my colleague which who I have work for four month on the graphic card implementation. His support was precious and our collaboration was efficient.

Many thanks to Rene Ranftl from Graz University of Technology who advised me on the Optical flow implementation.

2 Introduction

2.1 The company and the mission

The internship takes place in VITRONIC (Wiesbaden, Germany), a pioneer and one of the leading organizations worldwide in the field of machine vision. The company is specialized in four domains: Industry, Logistics, Traffic and Body scanning.

My internship deals with a theoretical background, which can be applied on different applications. I focus on two main problems: improving welds inspection in the automotive industry and rectifying the measurements of the VIPAC 3D scanner. In both case, the problems are similar and turn out to be classical optimization problems.

As my work is general and not specific to one of the company's domain, I do not work in a team. However, I work with another trainee, Stefan Gerning, in charge of filtering the data of the VIPAC scanner. This filtering step is also on optimization problem, which rely on the same theoretical background and can be parallelized on GPU.

Ingo Tzschichholtz, who is in charge of many projects in VITRONIC, supervises our work.

2.1.1 Weld inspection

The goal of the inspection is to detect any defect on a newly welded object. To achieve this task, it is necessary to apply a mask on the new object in order to extract its weld. Once the weld is extracted it can easily be compared with a reference weld to check its quality. The mask used on the process is obtained with a reference model on which the weld has been manually taken apart. The figure 2.1 shows a scanned object and the mask defined with a reference model.

The extreme temperatures resulting from the welding can deform the object. This is a problem to apply the mask since the reference model can differ from the new object. In addition, the scanner might not be accurate enough to guarantee that the new object will fit with the reference once, even without any deformation.

Hence, we must compute a function to go from the new object to the reference model. This function describes where each point of one object has to go in order to fit the other object. We can proceed as follow:

1. scan the new object,
2. compute the function which describes how to go from the new object to the reference model,
3. fit the mask to the new object,
4. extract the new weld and check its quality.

The difference between the two objects can be obtained by computing the transformation matrix, which describes the global rotation and translation to apply to one model to fit the other. One can then easily imagine a brut force algorithm:

- For each point of the new object, find the closest point on the reference model. This will result in n pairs of points, n being the number of point of the new object.
- Compute the transformation matrix with the n pairs of points,
- Apply the transformation to the new object,

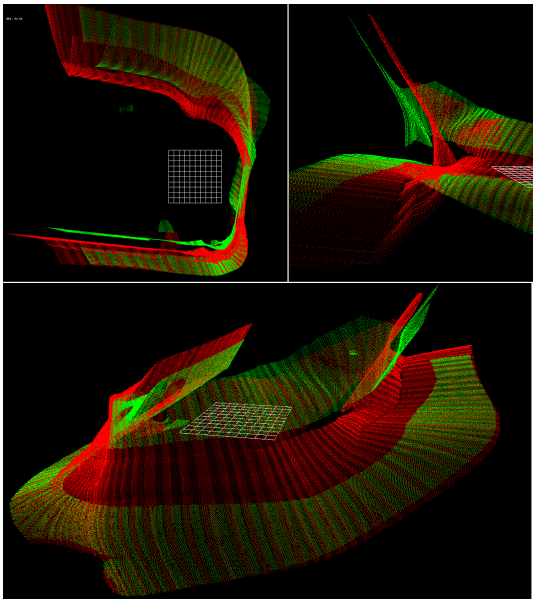

Figure 2.1: Mask (in green) and scanned object (in red).

- Iterate until the new object stops moving

However, this basic algorithm enables to define a global transformation for every point. This cannot describe how the object has been deformed when heated. Thus, the mask will not be correctly applied on the scanned object. One must use another technique to be more accurate.

The optical flow describes the transformation from one object to another at each point of the scene according to a regularization parameter. The lower the regularization is, the most independent the points can be, and the less the global shape of the object is taken into account. Since a parameter enables to give more weight to the global transformation or to the local transformations, the optical flow is a better solution of the problem.

2.1.2 VIPAC 3D scanner

This 3D scanner is in charge of measuring packages on a tray sorter. The measurements are then used to track packages and to estimate the number of packages that fit in a given volume. There is again some sources of imprecision:

- the trays are made of glued wood which cannot be accurately produced in a cheap way, i.e. two trays may differ slightly, even directly after production,
- trays wear out after awhile,
- their positions differ slightly since the shock absorbers in the tilt mechanism is made of deformable rubber sleeves.

In order to rectify the measurements, the VIPAC 3D scanner must know the position of the trays. Computing the optical flow between the current scanned tray and one reference tray enables to add an offset to the model and thus compute precise measurements.

3 Theoretical Background

3.1 Optical Flow with Total Variation methods

The choice of using optical flow to solve the previous problems is not only practical but it also goes along with a strategy. In fact, the optical flow algorithm is an optimization problem and so can be solve with mathematical tools which are convenient for any optimization problem. In this way, the theoretical background used to compute the optical flow can also be applied to a variety of optimization problem which goes from 3D scans filtering to image segmentation or 3D range integration.

In my bibliography I explained why total variation methods are the methods we chose to solve optimization problems and I proposed a solution to implement those methods. This section reminds the relevance of this technique for the optical flow and proposes directly a general solution and a pseudo-algorithm of the process in two dimensions.

3.1.1 Optical flow problems

There are two main problems when computing the optical flow: There is no information available on untextured regions, and it is only possible to compute the normal flow (the motion perpendicular to the edges). Hence, one must find a regularization to expand the displacement information to untextured areas and to create a coherent overall result.

3.1.2 Optical flow techniques

Horn and Schunck proposed a variationnal formulation of the optical flow problem [8]:

$$\min_{\mathbf{u}} \int_{\Omega} |\nabla u_1|^2 + |\nabla u_2|^2 d\Omega + \lambda \int_{\Omega} (I_1(\mathbf{x} + \mathbf{u}(\mathbf{x})) - I_0(\mathbf{x}))^2 d\Omega$$

I_0 and I_1 are the images, and $u = (u_1(x), u_2(x))^T$ is the two dimensional displacement field. The first tem is the regularization term and lead to smooth displacement fields. The second term is the data term. The free parameter λ weights between the data fidelity and the regularization term.

Because of the regularization term, this model does not allow for discontinuities in the displacement field, and does not handle outliers in the data term robustly.

A general solution using the total variation is proposed in [18]:

$$\min_{u} \{ \int_{\Omega} \psi(u, \nabla u, ...) d\Omega + \lambda \int_{\Omega} \phi(I_0(x) - I_1(x + u(x))) d\Omega \}$$

Where the functions ψ and ϕ depicts respectively the regularization and the data terms. Selecting $\phi(x) = x^2$ and $\psi(\nabla u) = |\nabla u|^2$ results in the Horn-Schunck model.

To preserve sharp motion boundaries, we can use a TV regularization as explained in my bibliography and in Thomas Pock PHD thesis [18]. In addition, we can allow outliers or occlusions with an L1 data penalty term. This leads to the choice of $\phi(x) = |x|$ and $\psi(\nabla u) = |\nabla u|$ which yields the following functional:

$$\min_u \{ \int_\Omega |\nabla u| d\Omega + \lambda \int_\Omega |I_0(x) - I_1(x + u(x))| d\Omega \}$$

Although this equation seems to be simple, it offers some computational difficulties since both the regularization term and the data term are not continuously differentiable.

3.1.3 Solution of the optical flow

Thomas Pock introduced a method in [18] to compute the N-dimensional optical flow estimation. In the solution, the image I_1 in the previous equation is linearized by using the first order Taylor approximation. This means that the whole algorithm needs to be embedded into an iterative warping approach to compensate for image non-linearities.

An N-dimensional diplacement map u is determined from two given N-dimensional images I_0 and I_1. The first order residual $\rho(u, u_0, x)$ with respect to a given disparity map u_0 is $I_1(x + u_0) + \langle \nabla I_1, u - u_0 \rangle - I_0(x)$. Additionally, u_d stands for the $d - th$ component of u ($d \epsilon \{1, ..., N\}$).

The solution now reads as:

$$\min_{u,v} \{ \int_\Omega \sum_d |\nabla u_d| d\Omega + \frac{1}{2\theta} \sum_d \int_\Omega (u_d - v_d)^2 d\Omega + \lambda \int_\Omega |\rho(v)| d\Omega \}$$

Minimizing this energy can be performed by alternating optimization steps:

1. For every d and fixed v_d, solve
$$\min_{u_d} \{ \int_\Omega |\nabla u_d| d\Omega + \frac{1}{2\theta} \int_\Omega (u_d - v_d)^2 d\Omega \}$$

2. For u being fixed, solve
$$\min_u \{ \frac{1}{2\theta} \sum_d \int_\Omega (u_d - v_d)^2 d\Omega + \lambda \int_\Omega |\rho(v)| d\Omega \}$$

The solution of the equation of the first step is given by:

$$u = v + \theta \nabla \cdot \mathbf{p}$$

where the dual variable $\mathbf{p}$ is obtained as the steady state of the projected gradient descend algorithm ($k = 1...K$) :

$$\tilde{\mathbf{p}}^{k+1} = \mathbf{p} + \frac{\tau}{\theta} [\nabla (\mathbf{v} + \theta \nabla \cdot \mathbf{p})]$$

$$\mathbf{p^{k+1}} = \frac{\tilde{\mathbf{p}}^{\mathbf{k+1}}}{\max\{\mathbf{1}, |\tilde{\mathbf{p}}^{\mathbf{k+1}}|\}}$$

where $\mathbf{p^0} = \mathbf{0}$ and the time step $\tau \leq 1/4$.

The solution of the minimization task in the equation of the second step is given by the following thresholding step:

$$v = u + \begin{cases} \lambda\theta I_1^x & if\ \rho(u) < -\lambda\theta(I_1^x)^2 \\ -\lambda\theta I_1^x & if\ \rho(u) > \lambda\theta(I_1^x)^2 \\ -\rho(u)/I_1^x & if\ |\rho(u)| \leq \lambda\theta(I_1^x)^2 \end{cases}$$

3.1.4 Implementation

Now that we have a good discretized solution of the optical flow problem, it is already possible to make a quick pseudo-algorithm in the 2D case ($N = 2$) with some pseudo-matlab code:

```
for k = 1:warps

    u0 = u;                          // initialize u0 with u

    I1warped = warp(I1,u0);                 // warp I1 with u0

    grad_I1warped = gradCentred(I1warped);       // compute the centred gradient of I1warped

    norm2_grad_I1warped = norm2(grad_I1warped);   // compute the L2 norm of grad_I1warped

    for j = 1:tvl1_iter

        for i = 1:rof_iter

            inc = v + theta * div( p );        // div( p ): computes the divergence of p

            p(:,:,:,1) = p(:,:,:,1)
                   + (tau/theta) * gradX(inc);  // gradX: computes the horizontal gradient
            p(:,:,:,2) = p(:,:,:,2)
                   + (tau/theta) * gradY(inc);  // gradY: computes the vertical gradient

            denom = max(1.0, norm2( p ));

            p(:,:,:,1) = p(:,:,:,1)./denom;
            p(:,:,:,2) = p(:,:,:,2)./denom;

        end

        u = v + theta * div( p );          // div( p ): computes the divergence of p

        ro_u = I1warped
           + scalar(grad_I1warped, u-u0) - I0;  // scalar(a,b): computes the scalar product between a and b

        where (ro_u < -lambda * theta * norm2_grad_I1warped)
          v = u + lambda * theta * grad_I1warped;

        where (ro_u > lambda * theta * norm2_grad_I1warped)
          v = u - lambda * theta * grad_I1warped;

        where (abs(ro_u) <= lambda * theta * norm2_grad_I1warped)
          and (norm2_grad_I1warped != 0)
          v = u - ro_u .* grad_I1warped ./ norm2_grad_I1warped;

        elsewhere
          v = u;

    end
end
```

The generally non-convex energy functional for optical flow becomes a convex minimization problem after linearization of the image intensities, but this linearization is only valid for small displacements.

In addition to the iterative warping approach, the energy minimization procedure must be embedded into a coarse-to-fine approach to avoid convergence to unfavorable local minima.

Image pyramids with a down-sampling factor of 2 are used for this purpose. Beginning with the coarsest level, the optical flow is computed at each level of the pyramid and the solution is propagated to the next finer level.

3.1.5 Conclusion

Thanks to the papers I read for my bibliography, I could understand perfectly the problems and the goals of my project. I was directly able to provide efficient solutions with interesting theoretical background. The total variation used in this method has many advantages (allows for discontinuities and gives information in untextured areas). Moreover, the theory behind the problem is adaptable to many computer vision problems.

Now that the theoretical background of the optical flow algorithm is defined, we must describe the environment in which it will be developed. The next section will be dedicated to OpenCL with which the computer vision algorithms will be parallelized.

3.2 OpenCL

The framework chosen to implement the parallelized optimization problems is OpenCL (Open Computing Language) since it can be processed on any device (CPUs and GPUs with AMD/ATI and Nvidia drivers), unlike CUDA which is an Nvidia specific framework.

The OpenCL framework combines a language (based on C99) for writing code that executes on OpenCL devices, and APIs that are used to define and control the platforms. It was designed to take full advantages of heterogeneous processing platforms consisting of CPUs, GPUs and others types of processors.

Using OpenCL, for example, a programmer can write general purpose programs that execute on GPUs without the need to map their algorithm onto a 3D graphics API such as OpenGL or DirectX.

3.2.1 The OpenCL architecture

The core ideas behind OpenCL can be described by the execution model and the memory model.

3.2.1.1 Execution model

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution.

Kernels The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item (a thread in the CUDA terminology, thread will refer to work-item in this report) and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item.

Work-items are organized into work-groups as shown in figure 3.1. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Global work-items are independent and cannot by synchronized. Synchronization is only allowed between the work-items in a work-group.

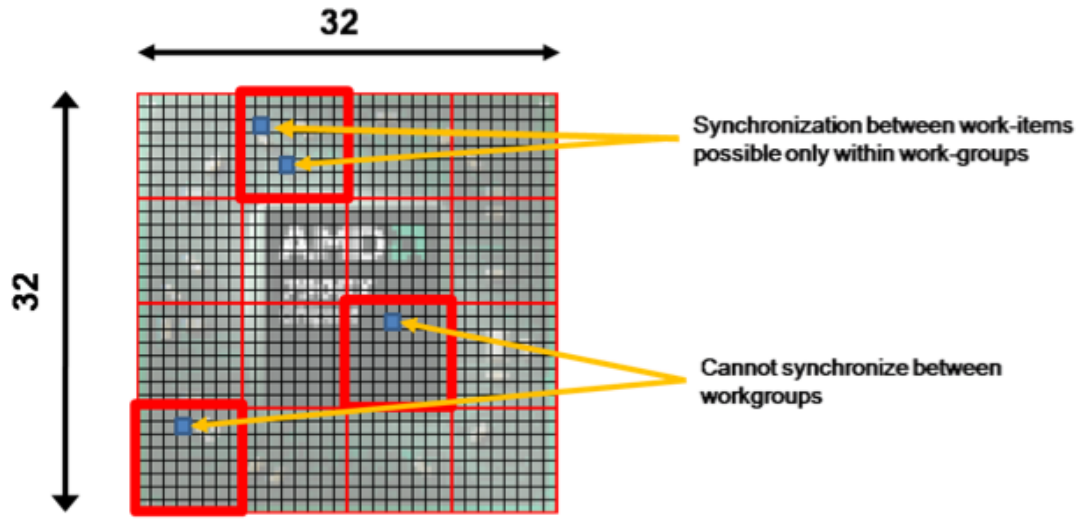

Figure 3.1: Grouping Work-items Into Work-groups

Figure 3.2 shows a two-dimensional image with a global size of 1024 (32x32). The index space is divided into 16 work-groups. The highlighted work-group has an ID of (3,1) and a local size of 64 (8x8). The highlighted work-item in the work- group has a local ID of (4,2), but can also be addressed by its global ID of (28,10).

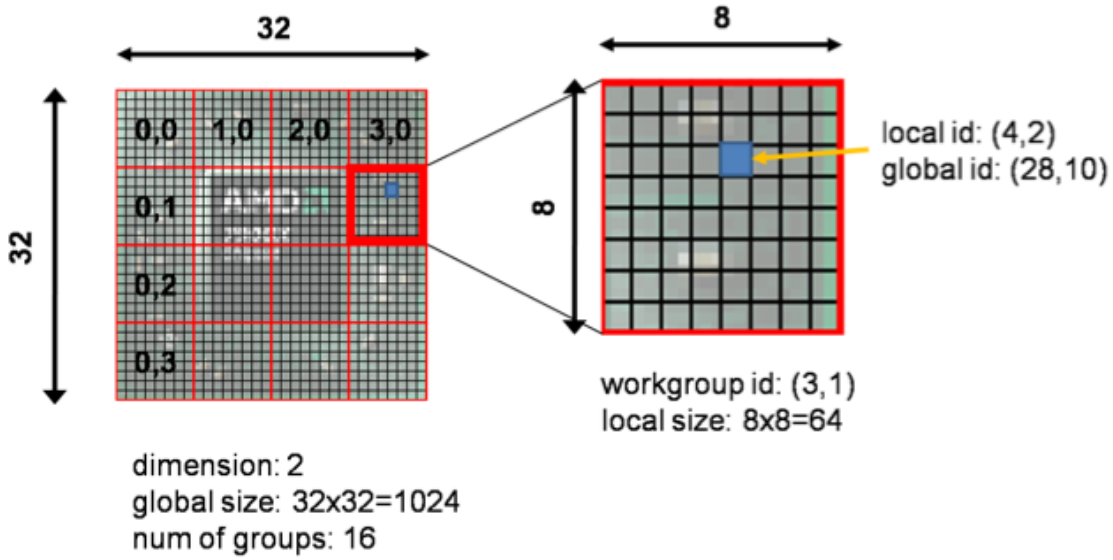

Figure 3.2: Work-group Example

The index space supported in OpenCL is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three.

The following example illustrates how a kernel in OpenCL is implemented. In this example, each element is squared in a linear array. Normally, a scalar function would be required with a simple for loop iterating through the elements in the array and then squaring it. The data-parallel approach is to read an element from the array in parallel, perform the operation in parallel, and write it to the output. Note that the code segment on the right does not have a for loop: It simply reads the index value for the particular kernel instance, performs the operation, and writes the output.

Table 3.1: Simple Example of Scalar Versus Parallel Implementation

Scalar C Function	Data-Parallel Function

`void square(int n,`
`  const float *a,`
`  float *result)`
`{`
`  int i;`
`  for (i=0; i<n; i++)`
`  result[i] = a[i]*a[i];`
`}`
	`kernel void square(`
	`__global const float *a,`
	`__global float *result)`
	`{`
	`  int id = get_global_id(0);`
	`  result[id] = a[id]*a[id];`
	`}`
	`// square executes over`
	`// "n" work-items`

Host Program

Context and Command Queues The host defines a context for the execution of the kernels. The context includes the following resources:

1. Devices: The collection of OpenCL devices to be used by the host.
2. Kernels: The OpenCL functions that run on OpenCL devices.
3. Program Objects: The program source and executable that implement the kernels.
4. Memory Objects: A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

The context is created and manipulated by the host using functions from the OpenCL API. After the context is created, command queues are created to manage execution of the kernels on the OpenCL devices that were associated with the context. Command queues accept three types of commands:

- Kernel execution commands: Execute a kernel on the processing elements of a device.
- Memory commands: Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.
- Synchronization commands: Constrain the order of execution of commands.

The command-queue schedules commands for execution on a device. These execute asynchronously between the host and the device.

3.2.1.2 The memory model

Work-item(s) executing a kernel have access to four distinct memory regions:

- Global Memory: a memory region in which all work-items and work-groups have read and write access on both the compute device and the host. This region of memory can be allocated only by the host during runtime.
- Constant Memory: a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.
- Local Memory: a region of memory used for data-sharing by work-items in a work- group. All work-items in the same work-group have both read and write access.

- Private Memory: a region that is accessible to only one work-item.

In most cases, host memory and compute device memory are independent of one another. Thus, memory management must be explicit to allow the sharing of data between the host and the compute device. This means that data must be explicitly moved from host memory to global memory to local memory and back. This process works by enqueuing read/write commands in the command queue.

Memory Objects Memory objects are categorized into two types: buffer objects, and image objects. A buffer object stores a one-dimensional collection of elements whereas an image object is used to store a two- or three- dimensional texture, frame-buffer or image.

The fundamental differences between a buffer and an image object are:

- Elements in a buffer are stored in sequential fashion and can be accessed using a pointer by a kernel executing on a device. Elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. Built-in functions are provided by the OpenCL C programming language to allow a kernel to read from or write to an image.

- For a buffer object, the data is stored in the same format as it is accessed by the kernel, but in the case of an image object the data format used to store the image elements may not be the same as the data format used inside the kernel. Image elements are always a 4- component vector (each component can be a float or signed/unsigned integer) in a kernel.

3.2.2 Example

A typical OpenCL application starts by querying the system for the availability of OpenCL devices. When devices have been identified, a context allows the creation of command queues, creation of programs and kernels, management of memory between host and OpenCL devices, and submission of kernels for execution.

The following example code illustrates the fundamentals of all OpenCL applications. It takes an input buffer (the data is considered arranged to represent a 2D image), transpose it, and stores the results in an output buffer, illustrating the relationship between device, context, program, kernel, command queue, and buffers.

The kernel code to transpose is simple and not optimized. An optimized version of this code can be up to 10 times faster.

```
__kernel void transpose(__global float * Iin, __global float * Iout, uint height, uint width)
{
  //global coordinates
  int gi = get_global_id(1);
  int gj = get_global_id(0);

  if(gi>=height || gj>=width)
    return;

  Iout[gj*height+gi] = Iin[gi*width+gj];
}
```

This code must end up in a char* (here *source*) with classical c++ functions.

Once the kernel is ready, one can begin a standard c++ application:

```
#include <stdio.h>
#include "CL/cl.h"

int main(void)
{
   // do anything in c++
```

One can then initialize an OpenCL environement by defining an OpenCL context, command queue and program.

```
    cl_context           context;
    cl_command_queue     commandQueue;
    cl_program           program;
    cl_kernel            kernel_transpose;
    cl_platform_id         platform;
    cl_uint              deviceCount;
    cl_device_id *         devices;
    size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
    size_t globalWorkSize[] = {roundUp(BLOCK_SIZE, width), roundUp(BLOCK_SIZE, height)};


  // [...]
    //Get the OpenCL platform
    err = clGetPlatformIDs(1, &platform, NULL);

  // [...]
    //Get the devices
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, deviceCount, devices, NULL);

    //Create the context
    context = clCreateContext(0, deviceCount, devices, NULL, NULL, &err);

  // [...]
    // create command queue
    commandQueue = clCreateCommandQueue(context, devices[0], CL_QUEUE_PROFILING_ENABLE, &err);

  // [...]
    // create the program
    program = clCreateProgramWithSource(context, 1, (const char **)&source, &program_length, &err);

  // build the program
    err = clBuildProgram(program, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
```

Now that the environment is initialized, one can create and run as many kernels as desired:

```
  // create the kernel
    kernel_transpose = clCreateKernel(program, "transpose", &err);

    // create the buffers/images, initialize them with data
    cl_mem b_Iin = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(float) *
        DATA_SIZE, data, &err);
    cl_mem b_Iin = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(float) *
        DATA_SIZE, data, &err);

    int j=0;
    // set the args values
    err  = clSetKernelArg(kernel_transpose, j++, sizeof(cl_mem), (void *) &b_Iin);
    err |= clSetKernelArg(kernel_transpose, j++, sizeof(cl_mem), (void *) &b_Iout);
    err |= clSetKernelArg(kernel_transpose, j++, sizeof(cl_uint), (void*) &height);
    err |= clSetKernelArg(kernel_transpose, j++, sizeof(cl_uint), (void*) &width);

  // enqueue the kernel command for execution
    err = clEnqueueNDRangeKernel(commandQueue, kernel_transpose, 2, 0, globalWorkSize, localWorkSize, 0, NULL
        , &GPUExecution);

  // wait for the end of execution
    err = clFlush(commandQueue);
    err = clFinish(commandQueue);

  // copy the data of the buffer/image into the host memory (uv)
  err = clEnqueueReadBuffer(commandQueue, b_Iout, CL_TRUE, 0, sizeof(float) * DATA_SIZE, data, 0, NULL, NULL)
      ;

    // release mem and event objects
    clReleaseMemObject(b_Iin);
    clReleaseMemObject(b_Iout);
    clReleaseEvent(GPUExecution);
```

After having executed the different kernels, one must not forget to release OpenCL objects:

```
// release program, kernel, context and command queue
clReleaseKernel(kernel_transpose);
clReleaseCommandQueue(commandQueue);
clReleaseProgram(program);
clReleaseContext(context);
```

4 Algorithms and Tools

4.1 Workflow

4.1.1 Discretizating the objects

The optical flow algorithm must be computed on a discrete space whereas the scanned objects are represented as point clouds. Thus, the first step is to discretize the two initial point clouds to create two 3D images consisting of white voxels where there are scan points, and black elsewhere. We have now two spaces, one continuous for the scanned object which has no boundaries, and one discrete for the images which goes from zero to image width, height and depth.

One could consider different methods to discretize the 3D objects. Instead of just setting the voxels to 1 where there are scan points, and 0 elsewhere, one could accumulate the values. That is to say one could add 0.2 to the voxel each time there is a scan point, thus creating a finer image. However, with a big resolution this technics implies low values per voxels. Since the blur averages those values, the result is a really smooth image which is not appropriate for the optical flow algorithm.

Another thing to consider is the scale value to go from the object space to the image space. A first method is to find the dimension in which the object is the largest, and compute a single scale value for every dimension. The result will be a non-cubic image with the proportion of the object respected. A second method is to define one scale value per dimension. This would stretch the object to have a cubic picture, in other words the pixels would not be squared. As the graphic card process the data in parallel, the size of the picture does not matter. However, it is always better to have the global dimensions of the 3D image as multiple of work-group dimensions. In this way the second discretization, which gives a fixed size cubic image, seems more appropriate.

4.1.2 Blurring the images

The algorithm is computed on each voxel separately. As a consequence some information (i.e. distance of the voxel to the closest edge) must be available on each voxel of both images. One good way to extend the information from the object's edges to its surrounding region is to blur the images. There are many ways to blur an image, the section 5.3 is dedicated to this step.

4.1.3 Computing the optical flow

The optical flow can then be computed in a coarse-to-fine approach to avoid convergence to unfavourable local minima. Beginning with the coarsest level, the optical flow is computed at each level of the pyramid and the solution is propagated to the next finer level.

The number of coarse levels depends on the difference between the two objects. The bigger the difference is, the higher the number of level must be. However, as the optical flow is not relevant anymore at very coarse levels, the quality of the result decrases when there are two many levels.

4.1.4 Applying the translations

Finally, the mask can be moved to the scanned object. This step consists in iterating through the vertices of the mask, converting their coordinates into the image space and applying the correct translation to them.

For better results it is possible to compute a linear interpolation between the optical flow values.

4.2 Optical flow algorithm

4.2.1 Step-by-step

Because the optical flow algorithm is not obvious to implement, I had a step-by-step approach of the problem:

- ROF: I started by implementing a 2D denoising algorithm on CPU using the ROF model. This first step gave me an overview of what the final algorithm would look like.
- TVL1: I modified the code to develop another 2D denoising algorithm for salt and pepper noise with the TVL1 model.
- Optical flow 1D: I could then implement the optical flow in one dimension, and then modify it to 2D and 3D, just as described in [18]. I could also implement the coarse-to-fine approach to test the algorithm efficiency on large movements.
- Optical flow 2D: I worked a lot on the 2D version of the algorithm since the 2D optical flow can be displayed with a handy color coding (hue = direction, saturation = magnitude), and the code is lighter than on the 3D version. Another advantage is that OpenCL and the graphic cards offer more possibilities in 2D than in 3D. This way I could develop some strategies to optimize the algorithm without having to care about the limitations due to 3D with OpenCL and the graphic cards.
- Optical flow 3D: Testing the 3D optical flow algorithm was challenging since I had to develop some tools to display 3D images and 3D vector fields.

In order to fully understand the algorithm and to make sure everything worked, I was in contact with René Ranftl, a colleague of Thomas Pock, the author of [18]. We could exchange some simple matlab code to make things clear, and this way I could implement the scale-space approaches [1] which were not explicitly explained in the thesis.

I started by implementing every algorithms in C++, then I switched to matlab, and finally I could parallelize the code with OpenCL.

One important thing I had to deal with, if not the most important, is to define the best strategies to implement and optimize the algorithms. This means implementing and comparing different versions of the same code. Thus, I had to deal with many different algorithms in 3 different languages and some with different versions. That is why I had to develop a flexible project organized to compare different versions of the code (ROF and TVL1, CPU vs. GPU, GPU with global memory vs. GPU with local memory, etc.).

I started with a simple global header which is included by both the OpenCL and the C++ code to define what algorithm and which version to compute. The CPU executes then the appropriate path to initialize the required memory for the GPU, and the GPU executes the appropriate kernels. The parameters of the different algorithms are also shared in this header. Then I developed a user interface with OpenGL and GLUI which enable to tweak the parameters to have a real time feedback of their influence. This interface can also be used to switch between different versions of the code, and end up being especially handy to compare CPU versus GPU performances.

4.2.2 Results

4.2.2.1 ROF, TVL1 and Optical Flow 1D

The results of the algorithms that I implemented before the final version optical flow are illustrated on figure A.5, A.6 and A.7 of the appendix.

The optical flow 1D computes the displacement between the two 2D pictures in the horizontal direction only, as described on figure A.7 in the appendix.

4.2.2.2 Optical Flow 2D

Figure A.8 of the appendix illustrates the result of the 2D optical flow algorithm in matlab. The optical flow is represented using the color coding (hue = direction, saturation = magnitude) taken from [18]. There is a local discontinuity at the bottom of the ellipse. This problem can be solved by slightly changing the parameters (λ, θ and τ) or by applying a median filter after each step of the coarse-to-fine pyramid. This is a good way to remove undesired outliers.

The next results concerns the CPU and GPU version of the 2D algorithm. The results will be presented in the following layout: the two left pictures are the initial images, the central images are the CPU results, and the images on the right are the GPU results.

The differences between the CPU and GPU results come from three things:

- the approximations of the GPU algorithms,
- the median filter of the GPU version,
- the linear behavior of the CPU version.

The block edges are not updated at every iterations, but rather used with approximated results and updated outside the kernel after a few iterations.

The median filter applied on the optical flow on the GPU version removes every outlier. Thus, the differences between the maxima and minima in the GPU optical flow are much lower. In this manner, the colors are much brighter on the GPU version.

The CPU version behaves in a linear manner. In fact, each pixel is processed one after the other. Hence the upper left pixels of the current pixel are already processed whereas the bottom right pixels are not. In the GPU version, the whole surrounding is not processed for the current pixel. In this fashion, the parameters (λ, θ and especially τ) do not have the exact same weight in each version of the algorithm.

First, I tried the algorithm on dense images like lena.pgm (figure A.9).

Yet, the goal was to apply the algorithm in 3D on point clouds. This is a different problem since the nature of the images is totally different. Thus, I started to try the algorithm robustness on different shapes. The simple shapes (not filled) as in figure A.10 do not provide enough information at each point to compute correct results. Even with the regulation introduced by the choosen solution, the algorithm needs more information about the overall shape at each point. In this manner, I tried to compute the algorithm on a textured shape (figure A.11). This test was not very convincing because it did not provide the correct local information.

The results were much better in the case of a filled shape, or a blurred shape as in figures A.13 and A.12.

Among my 2D tests, I tried the algorithm on different gradients and noticed that the algorithm works perfectly well in those case, as in figures A.14 and A.15.

Finally I developped the 3D version of the algorithm figure A.17.

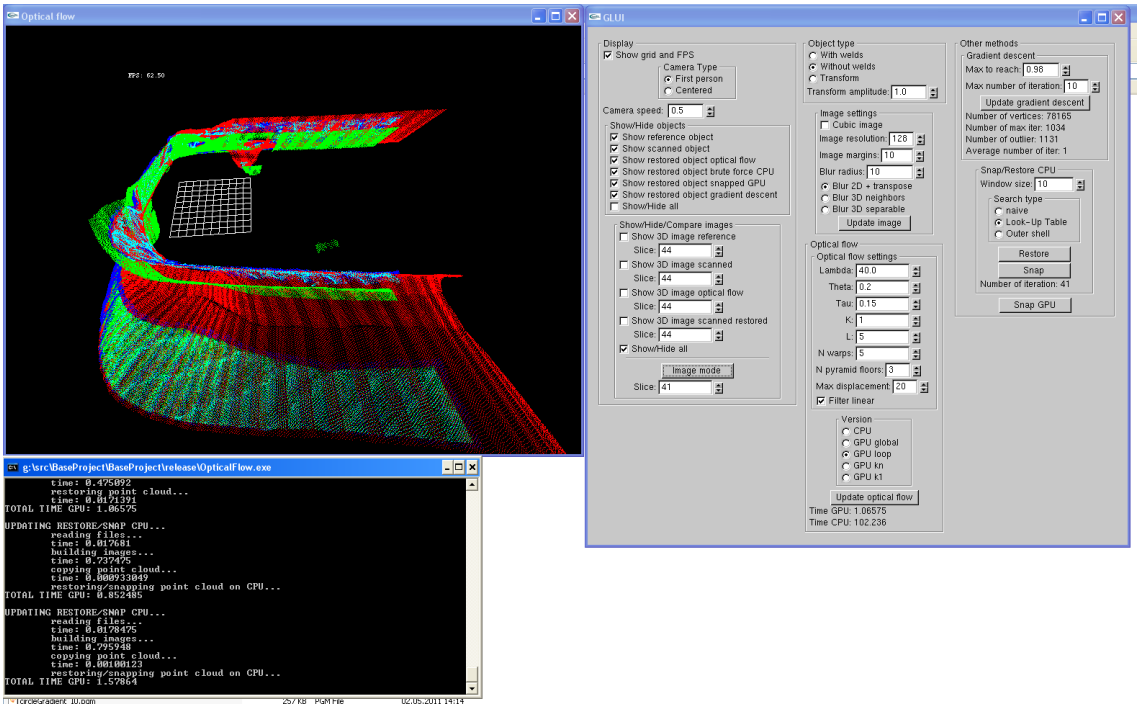

Figure 4.1: User interface in object mode

4.3 Tools

4.3.1 Reading and writing data

I implemented some tools to read and write images, but I also used the CImg library since it enables to view 2D and 3D images during execution. I still had to implement methods to view more than one 3D image at a time in order to compare different results. I also used the internal library and format of VITRONIC to read and write 3D objects.

4.3.2 Visualizing data with OpenGL and GLUI

To visualize the 3D data, I started by implementing an OpenGL environment with a standard camera which enables to choose which object to display. Then I designed a Graphical User Interface (GUI) with the GLUI library to be able to see the effect of parameters in real time, and to visualize the 3D images and the 3D objects in the same time. Each parameter can be set with a spinner and the optical flow can be updated with a simple button. I extended the GUI to compare and display most of the algorithms that I developed.

The 3D images are also updated and displayed in slices. The slices are displayed as textures on simple OpenGL quads. Some spinners on the UI determines which slice to display and where. In fact, each slice is positioned on the corresponding 3D object (point clouds), and the z position changes according to the number of the slice, as shown on figure A.4.

A special mode enables to compare the 3D images. In this mode, the images are moved side to side, one global spinner affects the number of slice for every image, and the camera moves to the best point of view to compare the images.

The final program is composed of three windows as described on figures 4.1 and 4.2. The first contains the OpenGL scene with objects (point clouds) and their corresponding images. The second, made with GLUI, contains every buttons to control the algorithms and the scene. The third window is the console window which displays information about the algorithm in process and the computation speeds.

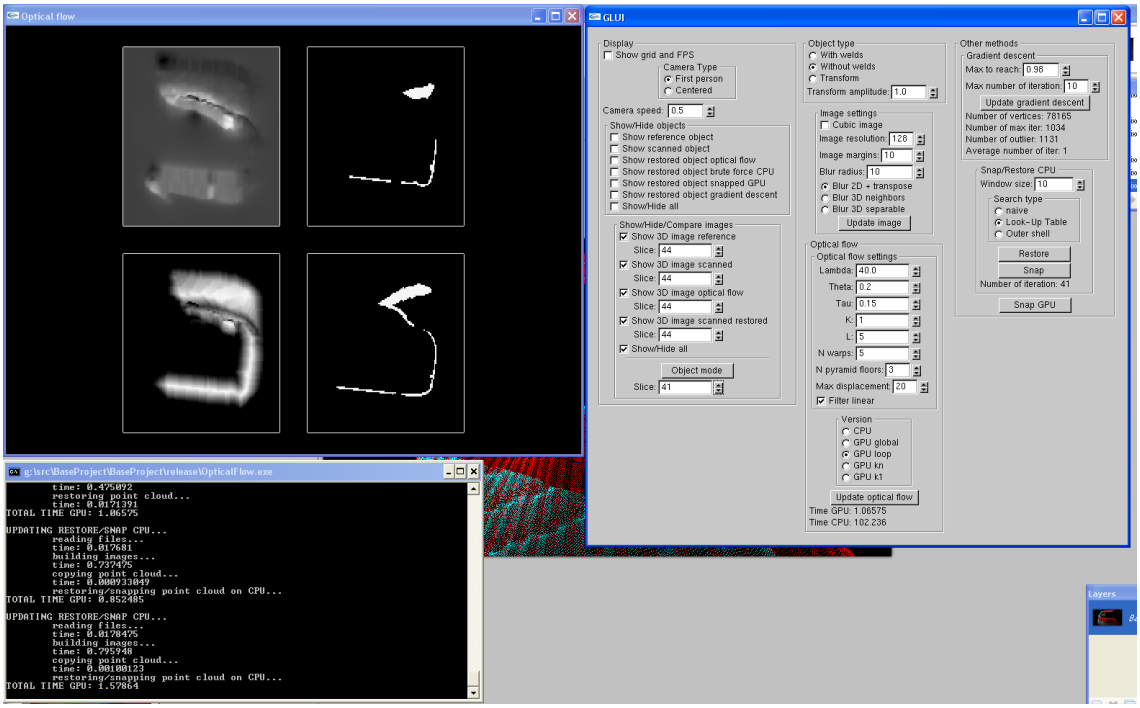

Figure 4.2: User interface in image mode

Figures A.1, A.2 and A.3 of the appendix describe the user interface and its fonctionnalities.

4.3.3 Optical flow visualizing with color coding

Finally, I developed the color-coding using the HSV (Hue, Saturation and Value) color model where the color hue (angle form 0 to 360 degree) is used to display the direction of the translation and the saturation for its magnitude. The dx and dy components of the translation have to be converted into an angle:

$$hue = 180 + 360 * \frac{atan2(dx, dy)}{2\pi}$$
$$saturation = \sqrt{dx * dx + dy * dy}$$

In the 3D case, it is also possible to give the dz coordinate to the color value:

$$value = 1 - dz$$

Then, the HSV color must be converted in RGB, one common method is described in [20]: Given a color with hue $H \in [0°, 360°]$, saturation $S_{HSV} \in [0, 1]$, and value $V \in [0, 1]$, we first find chroma:

$$C = V * S_{HSV}$$

Then we can find a point (R1,G1,B1) along the bottom of the faces of the RGB cube, with the same hue chroma as our color (using the intermediate value X or the second largest component of this color):

$$H' = \frac{H}{60^\circ}$$
$$X = C(1 - |H' mod 2 - 1|)$$

$$(R_1, G_1, B_1) = \begin{cases} (0,0,0) & if\ H\ is\ undefined \\ (C,X,0) & if\ 0 < H' < 1 \\ (X,C,0) & if\ 1 < H' < 2 \\ (0,C,X) & if\ 2 < H' < 3 \\ (0,X,C) & if\ 3 < H' < 4 \\ (X,0,C) & if\ 4 < H' < 5 \\ (C,0,X) & if\ 5 < H' < 6 \end{cases}$$

Finally, we can find R, G and B by adding the same amount to each component, to match value:

$$m = V - C$$
$$(R, G, B) = (R_1 + m, G_1 + m, B_1 + m)$$

5 Optimization and Strategies

5.1 Optimization of the parameter setting

5.1.1 Different parameters

There are numerous parameters for the optical flow algorithm, those parameters must be set to get the optimal results and performances.

OpenCL device related parameters The workgroup sizes are OpenCL specific and highly depend on the OpenCL version and OpenCL devices on which the algorithms are computed. The sizes should be multiples of 16 to avoid bank conflicts when reading and writing memory and get optimal performances. Nevertheless, the devices have a maximum number of register per block (workgroup)
$CL\_DEVICE\_REGISTERS\_PER\_BLOCK\_NV$.

The number of thread (work-item) in a block (which is the multiple of the size of the block in each dimension: $BLOCK\_WIDTH * BLOCK\_HEIGHT * BLOCK\_DEPTH$ in 3D) times the number of register required by a kernel must be lower or equal than this maximum number of register per block.

$BLOCK\_WIDTH * BLOCK\_HEIGHT * BLOCK\_DEPTH *$
$N\_REG\_REQ\_BY\_KERNEL <= CL\_DEVICE\_REGISTERS\_PER\_BLOCK\_NV$

In other words, the optimal sizes are the biggest multiple of 16 which once multiplied by the number of required register is lower than the maximum number of register supported by the OpenCL device. Since the kernels run with different number of dimension (1D, 2D or 3D) the number of thread used varies a lot. Furthermore, some kernels require having the same size for each dimension and others run faster if the sizes are different. As a consequence, the sizes of workgroups depend on the kernels.

Total variation parameters In section 3.1.3 we introduced the mathematical parameters λ, τ and θ. Although the best value of τ is easy to find and do not highly depend on the two other parameters, finding the optimal combination of λ and θ appears to be more complex.

There are also some algorithmic parameters which define the number of loop iterations:

- K denotes the iteration number of the projected gradient descent algorithm,
- L and N_WARPS are the overall iterations number, L is for the inner loop (actual computation of the optical flow), and N_WARPS the outer loop which includes the warp update and the synchronization between the variables u_0, u, and v,
- $N\_PYRAMID\_FLOOR$ describes the number of sample to work with.

The $MAX\_DISPLACEMENT$ parameter is used as a threshold to filter the extreme values. The median filter algorithm used in the process requires minimum and maximum bounds set to - and + $MAX\_DISPLACEMENT$. The values of the optical flow can be clamped between those two values before the filter is applied. Finally, $BLUR\_RADIUS$ denotes the size of the blur applied to the images. It influences the precision of the optical flow and the maximum scale of the detected movements.

Image resolution and margins Two more parameters define the 3D images resolution and their margins. The resolution is of high importance since it determines the results precisions and has a drastic effect on performances. Since we use a coarse-to-fine approach, a high resolution will not be affected by strong displacements, but the speed is extremely sensitive to the image size. The margins are used to take into account the whole blurred image (bigger than the original image by two $BLUR\_RADIUS$).

5.1.2 Optimizing parameters

There are 17 parameters on the overall. It is theoretically possible to design an optimization problem of dimension 17 to find the best combination of those parameters for a given optical flow problem. In fact, the overall problem can be considered as a function of those parameters returning the error between the computed optical flow and the real one, the goal is to minimize it. There are numerous ways to solve such optimization problems, like Newton algorithms or genetic algorithms. However, to compute an error between the result and the real difference is tricky. Some results can fit very well in one precise region of the object and not on the others, other results could fit correctly on the overall but be really noisy around the object. Furthermore, many parameters are independent of others and experience is often sufficient to find the best values. The result is really sensitive to some parameters like θ and λ and not to others.

5.1.2.1 A simple grid

I choose to begin with a simple method to find the global minima easily and to develop something more complex and powerful in the future if necessary. The simple solution consists of a grid. For each parameter we compute the optical flow in a range of predefined values. For example, we can run $n * m$ times the optical flow with θ going from 0.01 to 0.11 in n step and λ from 1 to 101 in m step.

Several improvements could be done, the first one would be to increase the step size exponentially (or in a non linear way) since the algorithm is sensitive for low values of parameters but not when the values are high.

A second improvement would be to recompute a finer grid with the first results. The minima of this first coarse grid indicate the precise range in which the minimum is likely to be.

To keep the algorithm simple, the finer grid must be set manually, an automated precise version would be complex to implement (the new grids must fit the best all global minima). A more classical approach can be used to develop an automated version of the research of parameters.

Figure 5.1: A few grids obtained with this technic. Dark pixels mean low errors.

5.1.2.2 User interface

The user interface used together with the grid enabled to easily optimize the results. In fact, the UI gives real time feed back of the effects of each parameter. This is a really powerful tool since it is extremely simple to use and the code does not change in anyway. Furthermore, there is no need to compute the optical flow error. The user can directly see if the mask fits better the scanned object and how. In fact, the results can be better because the optical flow takes the local features of the objects into account, or because the mask is more stretched but fit closer to the scan. The grid method does not allow making a difference between the types of results.

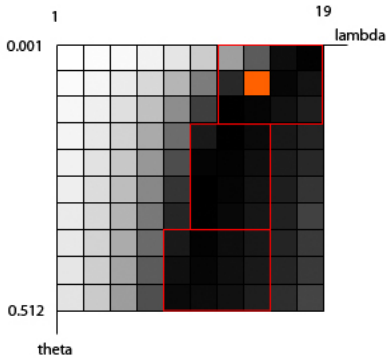

Figure 5.2: Non linear grid (theta from 0.001 to 0.512 and lambda from 1 to 19) of the optical flow errors. The red rectangles are the areas where the next finer grids should be initialized. The orange rectangle is the current global minimum.

5.2 Strategies and optimization with respect to the hardware and execution speed

Once the optical flow algorithm implemented and integrated in the workflow, making some strategic choices can optimize the overall program. Those strategies concern mostly the GPU architecture or its limitations.

5.2.1 Number of registers

As we have seen in section 5.1.1 the number of register is limited by the OpenCL devices (the graphic card in our case).

There are some special functions and extensions to display the number of registers used by a kernel, but those functions do not execute whenever the compiler has some previous data in memory. Therefore, a script is used to delete obsolete data at each run in order to always get information about registers.

One possibility to limit the number of register used by a kernel is to force it to a number with the compilation option "-cl-nv-maxrregcount=N" where N can be any positive number. This technic can be used to reduce slightly the register number to obtain optimal computation speed. However, when the number of required register is too large compare to "maxrregcount", the kernel compiles but does not execute or can have unexpected and random behaviours. It does not even return an error. The only ways to know if the kernel was correctly executed is to see if the results are consistent or to check the computation speed (non executed kernels are obviously very fast). Since there is no way to know for sure if the kernels were correctly executed, it is better not to use this compilation option.

A better approach would be to limit the number of register manually. This is sometimes possible by rewriting the code, for example by splitting complex lines into many small lines with intermediate variables. It is also possible to redesign the architecture to gain some registers, for instance using image objects requires a lot more registers than simple buffer objects. Packing memory can also help reducing the number of registers.

One might also consider removing the loops from the kernels and putting them outside as explained on figure 5.3 (calling the kernel in the loop on the host side, instead of running the loop inside the kernel).

One general and drastic method is to split big kernels into smaller ones and execute them one after the other.

Reducing the number of register is not only important to run the kernels, but can also be critical to set the workgroup size as multiple of 16 to avoid bank conflicts; as seen in subsection 5.1.1.

```
    // Host side:

    // This loop was inside the kernel but can be placed here on the host side
  for(int n=0 ; n<MAX_ITER ; n++)
  {
        // enqueue the kernel command for execution and wait for the end of execution
          err = clEnqueueNDRangeKernel(commandQueue, kernelNoLoop, 2, 0,       globalWorkSize, localWorkSize,
              0, NULL, &GPUExecution);
          err = clFlush(commandQueue);
          err = clFinish(commandQueue);
    }
```

```
kernel void kernelNoLoop(float * I, uint height, uint width)
{
     int i = get_global_id(1);
     int j = get_global_id(0);

 //     This loop can be placed outside the kernel to reduce the number of register
 //     and to synchronize work-items
 //     for(int n=0 ; n<MAX_ITER ; n++)
 //     {
           // kernel code, do anything
 //    }
}
```

Figure 5.3: Program with a for loop removed from the kernel and placed on the host side.

5.2.2 Thread synchronization

Local memory is much faster than global memory. However, it is not possible to synchronize two different work-groups during the execution of a kernel. Thus, if each work-item needs its neighbors for the algorithm, a loop inside the kernel leads to synchronization problems. In fact, work-items on work-group edges can not be updated correctly since they do not have access to their neighbors. Thus, the programmer should decide if he approximates the results with small loops inside the kernel and no synchronization, or if he synchronizes the work-items outside the kernel as described on figure 5.3.

5.2.3 Memory management

5.2.3.1 Choosing the appropriate memory

The choice of memory used to compute the algorithm is one of the most important steps in the development process. Indeed, the computations can easily be speed up by a factor of ten when using the appropriate memory.

The diagram on figure 5.4 describes the whole procedure to choose the appropriate memory. The numbers in purple circles refer to the following comments:

1. The first thing that determines the choice is if the memory is an input or output, or if it is internal to the kernel. One should be aware that synchronization between work-groups is not possible, therefore global memory can be necessary even in case of an internal memory.

2. Even though the memory is an input or output, copying it to local or private memory can significantly speed up the kernel when there are many memory reads or writes.

3. The choice depends obviously on the data type, if it is an image or a simple array.

4. OpenCL C language provides built-in functions to read from or write to an image, manage one or four channels easily, check boundary conditions and most important compute 2D and 3D linear interpolations.

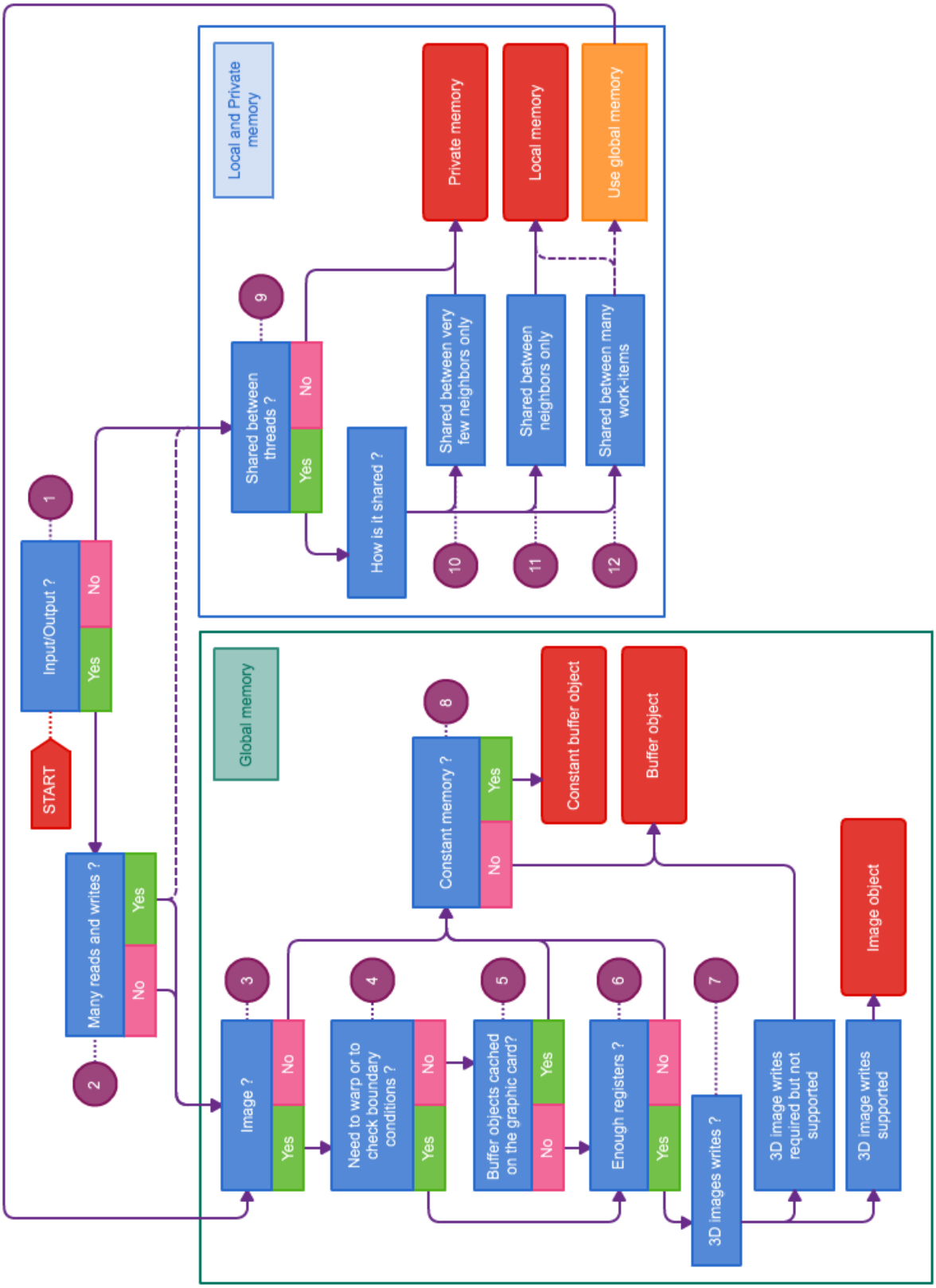

Figure 5.4: Diagram to help choosing the appropriate memory.

5. Latest graphic cards cache image and buffers objects whereas older ones cache only image objects.

6. Using image objects requires more registers since the OpenCL C built-in functions invoke more complex code than classic addressing.

7. On most OpenCL devices, even the latest Nvidia graphic cards, 3D image writes are still not supported. For this reason, buffer objects are still of great use. Note that since they are cached on recent graphic cards, there is no significant performance difference anymore.

8. It is possible to define buffer objects as constant. Although every buffer objects is cached on recent graphic cards, this is only true for constant buffers on older graphic cards.

9. One of the most important thing to consider when designing a kernel is thread synchronization. It is possible to synchronize work-items only within work-groups, not between different work-groups.

10. When the work-items need only a few neighbors, it is possible to create one private variable for each neighbor from global memory.

11. Local memory is extremely useful to share data within work-groups. This is a good compromise between private memory which cannot be shared and global memory which is slow to read, write and transfer from the host side.

12. Global synchronization must be executed outside the kernel, this might mean to exclude a loop from a kernel. A loop which could be inside a kernel can be moved outside (on the host side) over the kernel call. In this case, the memory must be copied to the global memory in order to remain between the kernel calls.

The choice of memory highly depends on the type of algorithm. Although the diagram is extremely usefull to understand the different strategies behind parallel programming, there is no easy way to obtain the best solution. One must try different implementations and select the best one.

5.2.3.2 GPU and CPU transfers

Applications should strive to minimize data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels with low parallelism computations. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory. In the case of the optical flow computation, the up and down sampling are not faster on GPU if we consider memory transfer, but implementing those operation on the graphic card enable to avoid memory update from the host to the device. For example, the image textures used for the objects stays on the card during the whole process (down sampling and optical flow). In the same way, the buffers of the variables u and v which store the optical flow can be reused on every smaller resolution of the pyramid, there is no need to create two buffers for each floor.

The memory management of the optical flow algorithm is organized as follow:

- blur both 3D images on buffer objects,

- create every image objects for the images and their down sampled version,

- create the buffers u and v for the optical flow of the largest size, those will be reused for each floor (only a small portion of the buffers will be used to compute the first floors),

- compute the optical flow for each floor, and upsample the result each time to reuse it in the finer resolution.

5.2.3.3 Packing memory

One important thing to consider is the image (or buffer) creation time. To avoid creating many images, it is possible to pack different images into a single one since images have four components by default (RGBA). For example, in the 2D optical flow case, the variables u_x, u_y, v_x and v_y can be packed together in a single RGBA image. This considerably reduces the image creation time.

Packing memory can also be used to avoid bank conflicts, this can lead to significant performance improvements. The block of memory accessed together must be placed together so that consecutive work-items access memory from the same bank.

5.3 Choosing the appropriate strategy

There are many possibilities to blur images on a GPU, some techniques are efficient, others are not. We will describe many techniques to illustrate the different choices which must be done when designing parallel programs.

5.3.1 Blur technics

We will focus on two approaches to blur a picture on CPU, and then we will see six different ways to adapt them on GPU.

The blur we will describe is a very simple blur consisting of iteratively spreading the values of pixels to their neighbors, as described on figure 5.5.

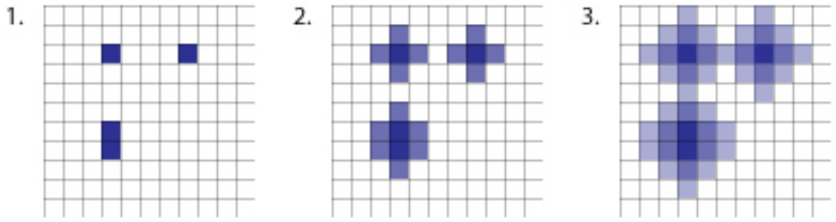

Figure 5.5: Naive blur

Here is the first naive solution we will consider:

```
void blur(float * I, uint blurRadius, uint height, uint width)
{
  float alpha = 1.0/(float)blurRadius;
  for(int n=0 ; n<blurRadius ; n++)
    for(int i=0 ; i<height ; i++)
      for(int j=0 ; j<width ; j++)
      {
        float pmax = I[(i-1)*width+j];
        if(I[(i+1)*width+j]>pmax)
          pmax = I[(i+1)*width+j];
        if(I[i*width+j+1]>pmax)
          pmax = I[i*width+j+1];
        if(I[i*width+j-1]>pmax)
          pmax = I[i*width+j-1];

        if(I[i*width+j]<pmax-alpha)
          I[i*width+j] = pmax-alpha;
      }
}
```

This algorithm is very slow since it implies blurRadius*(height*width) iterations, but it is a good example to introduce problems due to the GPUs architectures because each pixel needs only its closest neighbors.

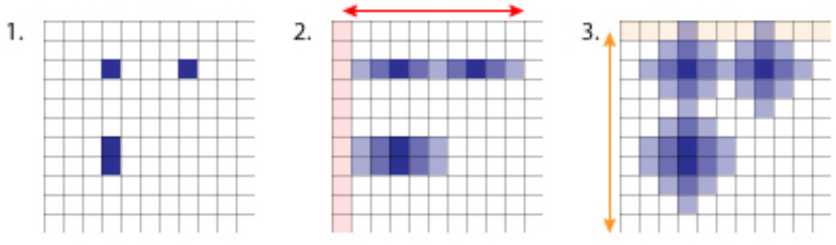

Figure 5.6: Blur algorithm which uses the separable property

The figure 5.6 illustrates the second version of the blur which takes advantage of the blur's linearly separable property by dividing the process into two passes.

This is the code of the second versions:

```
void blur(float * I, uint blurRadius, uint height, uint width)
{
  float alpha = 1.0/(float)blurRadius;
  for(int i=0 ; i<height ; i++)
  {
    for(int j=0 ; j<width-1 ; j++)
    {
      float p = I[i*width+j]-alpha;
      if(I[i*width+j+1]<p)
        I[i*width+j+1] = p;
    }
    for(int j=width-1 ; j>0 ; j--)
    {
      float p = I[i*width+j]-alpha;
      if(I[i*width+j-1]<p)
        I[i*width+j-1] = p;
    }
  }
  for(int j=0 ; j<width ; j++)
  {
    for(int i=0 ; i<height-1 ; i++)
    {
      float p = I[i*width+j]-alpha;
      if(I[(i+1)*width+j]<p)
        I[(i+1)*width+j] = p;
    }
    for(int i=height-1 ; i>0 ; i--)
    {
      float p = I[i*width+j]-alpha;
      if(I[(i-1)*width+j]<p)
        I[(i-1)*width+j] = p;
    }
  }
}
```

This version is much faster since it iterates through each pixel at maximum 4 times, whereas the first version iterates blurRadius times.

5.3.2 Implementing the blur on GPU

5.3.2.1 2nd algorithm with global memory

One obvious solution would be to apply the second algorithm on a 2D range. Every pixel would be parallelized but only the work-items of the left and top edges would process the image. The work items of the left edge would process the front and back passes in the x direction, and the top edge in the y direction. The process would be exactly as on figure 5.6. This implies that every other work-items would be created but not used by the GPU, which make it a very slow process.

The goal is now to have more work-items processing the image.

5.3.2.2 1st algorithm with global memory

It is also possible to implement the first blur algorithm on GPU with global memory:

```
kernel void blurGlobal( __global float *I, uint blurRadius, uint width, uint height)
{
    int i = get_global_id(1);
    int j = get_global_id(0);

    float alpha = 1.0/(float)blurRadius;

    float temp = I[(i-1)*width+j];
  if(I[(i+1)*width+j]>pmax)
    pmax = I[(i+1)*width+j];
  if(I[i*width+j+1]>pmax)
    pmax = I[i*width+j+1];
  if(I[i*width+j-1]>pmax)
    pmax = I[i*width+j-1];

  if(I[i*width+j]<pmax-alpha)
    I[i*width+j] = pmax-alpha;
}
```

Then, this kernel can be called in a loop to run it *blurRadius* times. This method seems simple and efficient. However, it uses global memory which is much slower than local memory. Using local memory can drastically improve performances, especially when the number of memory reads and writes is important (which is not the case here, this is just an example).

5.3.2.3 1st algorithm with local memory

It is only possible to access local memory within work-groups. Work-items on the work-group edges must access their neighbors through global memory. One way to implement this is to allocate a block of local memory two pixels wider and higher than the work groups. Hence, one can initialize the whole block of local memory with a block of global memory plus the surrounding edges as shown in the figure 5.7.

The local memory is allocated on the host side. The kernel is described on figure 5.8.

In this example there are not many memory reads and writes; yet the process could be more complex and require some loops as in the optical flow algorithm. In such cases, this solution is much faster than global memory solutions. This is the solution which I implemented for the optical flow.

Synchronization problem Note that the kernel takes two images, one input which will be read only, and an output which will be written. In fact, there is no way to synchronize the work-groups, that means one work-group can have finish the execution

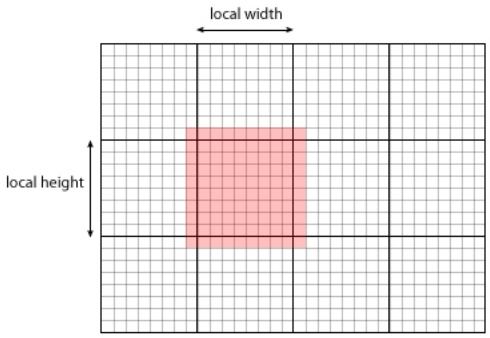

Figure 5.7: local memory initialization: the red square represents the local memory of size (local width+2)*(local height+2), the grid represents the work-items in their work-groups.

whereas its neighbors would not have start. In other words, it is not possible to stop the surroundings work-groups to make sure we do not initialize the local memory with the picture already blurred.

On figure 5.9 we can see synchronizations problem on a simple kernel which computes the gradients with only one buffer. The vertical white lines which appear randomly on the work-groups edges are the result of computing the gradient with the already calculated gradient of the neighbor work-groups.

Because of this synchronization problem, it is not possible to loop over the process inside the kernel without approximating the results. The loop which iterates $blurRadius$ times must be outside the kernel, on the host side on the kernel call. The local memory must be transferred to a global memory so that the data remains between two kernel calls.

In order to improve performances, one must find a way to put this loop inside the kernel. There are two solutions for this problem.

5.3.2.4 Solving synchronization problem with edge duplications

The first method is to duplicate the edges around the initial work-groups (blocks) n times to create a larger image, n depending on the number of iteration of the loop ($blurRadius$ in our case). The figure 5.10 illustrates the procedure. In this manner, the final work-groups have all the information required to run independently of the others and the work-groups cores will not be approximated. However, the major drawback of this method is that it requires high redundancy. In 2D, an image of size $width * height$ with blocks of size $blockWidth * blockHeight$ will have a size of $(width + n * 2 * width/blockWidth) * (height + n * 2 * height/blockHeight)$. This is a huge increase of memory especially for a large number of iterations n. Yet, the biggest drawback is the work-group size since the number of registers on the graphic card limits it. The work-group width and height must each be at least 4 times the number of iterations. This makes the solution impossible to implement for 3D applications which require more than very few iterations.

5.3.2.5 Solving synchronization problem with offsets

The second method conists in executing the kernel n times, each time with a different offset. It is possible to implement both algorithms with this method. The figure 5.11 illustrates the solution.

5.3.2.6 2nd algorithm with 1D blurs

The last solution to implement the blur on GPU is to use the second algorithm in 1D with two kernels. The first kernel blurs in 1D, meaning $imageWidth$ work-items will each iterate through their respective column. Then the second kernel transposes the image. Finally, the first kernel runs again on $imageHeight$ work-items.

```
kernel void blurGlobal( __global float *Iin,  __global float *Iout,  __local float *Ilocal, uint blurRadius,
    uint width, uint height)
{
  //global coordinates
    int gi = get_global_id(1);
    int gj = get_global_id(0);

  //local coordinates
    int li = get_local_id(1)+1;
    int lj = get_local_id(0)+1;

  //local size
    int lHeight = get_local_size(1)+2;
    int lWidth = get_local_size(0)+2;

    float alpha = 1.0/(float)blurRadius;

    // local memory initialization on every work-items
    Ilocal[li*lWidth+lj] = Iin[gi*width+gj];

    //local memory edges initialization
    if(li-1==0)
      Ilocal[(li-1)*lWidth+lj] = Iin[(gi-1)*width+gj];
    if(li+1==lHeight-1)
      Ilocal[(li+1)*lWidth+lj] = Iin[(gi+1)*width+gj];

    if(lj-1==0)
      Ilocal[li*lWidth+lj-1] = Iin[gi*width+gj-1];
    if(lj+1==lWidth-1)
      Ilocal[li*lWidth+lj+1] = Iin[gi*width+gj+1];

    float p = Ilocal[i*width+j]-alpha;

  // blur with local memory
    float pmax = ILocal[(li-1)*lWidth+lj];
  if(ILocal[(li+1)*lWidth+j]>pmax)
    pmax = ILocal[(li+1)*lWidth+lj];
  if(ILocal[li*lWidth+lj+1]>pmax)
    pmax = ILocal[li*lWidth+lj+1];
  if(ILocal[li*lWidth+lj-1]>pmax)
    pmax = ILocal[li*lWidth+lj-1];

  if(ILocal[li*lWidth+lj]<pmax-alpha)
    Iout[i*width+j] = pmax-alpha;
}
```

Figure 5.8: Naive blur using local memory

Figure 5.9: Gradient of lena.pgm calculated on GPU with only one input/output buffer. The white edges are computed with the already calculated gradient of lena.

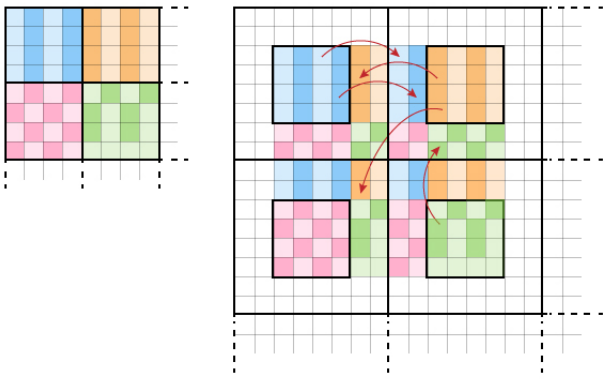

Figure 5.10: Work-group edge duplications. The original image (left) is divided into blocks. The resulting image (right) consists of the same blocks with duplicated edges. The work-groups size will be the same as the block size plus the edges.

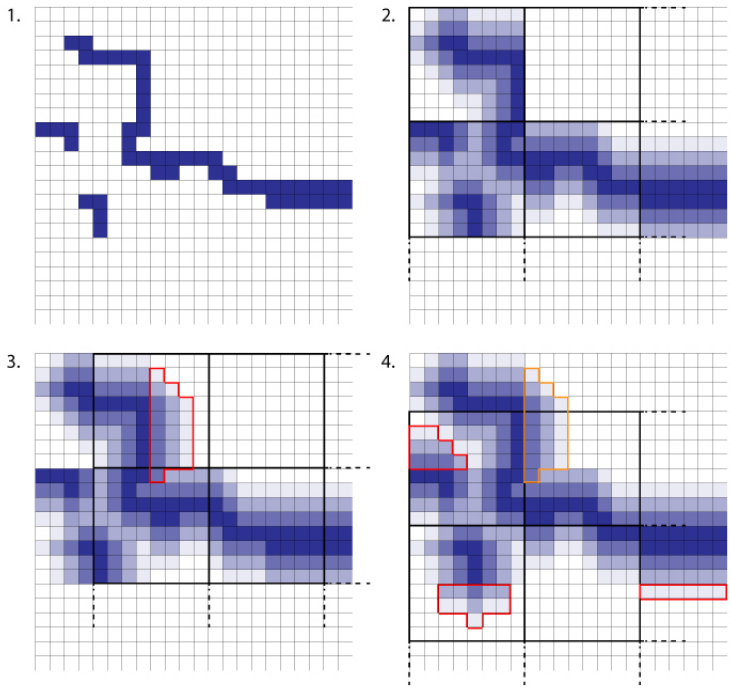

Figure 5.11: Runing the kernel with different offsets to solve synchronization problems. Each work-group computes the image independently of the others (step 2.). The process is computed again with a different offset (step 3. and 4.) to update the areas which were not taken into account.

I had to implement the blur algorithm in 3D to prepare the images for the optical flow. The process is then a 7 step procedure as explained on figure 5.12:

1. Blur the 3D image in the z dimension (call a 2D range kernel and iterate through the z dimension),
2. Transpose the x and z dimensions (3D kernel),
3. Bur in the z dimension,
4. Transpose the z and y dimensions,
5. Bur in the z dimension,
6. Transpose the z and y dimensions,
7. Transpose the x and y dimensions,

5.3.2.7 Results

I implemented those algorithms to blur a 128 wide 3D image with a blur radius of 10 voxels. I obtained the following results:

- 2nd algorithm with global memory: 3.881 seconds,
- 1st algorithm with local memory: 0.381 seconds,
- 2nd algorithm with local memory: 0.147 seconds,

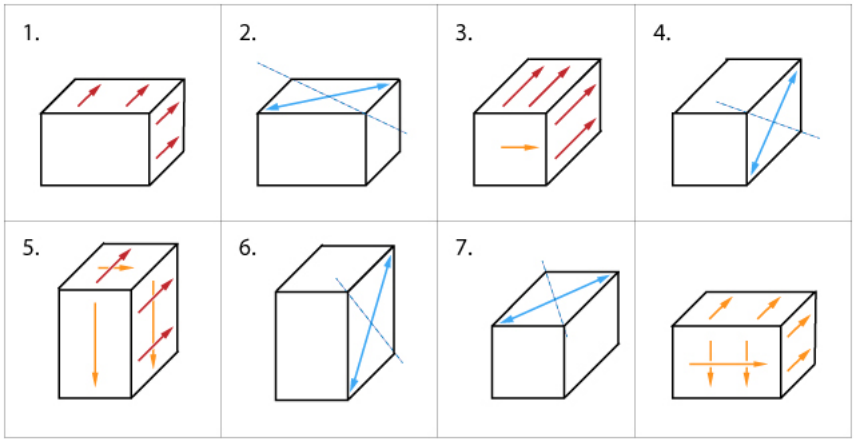

Figure 5.12: 7 step process of a 3D blur.

- 2nd algorithm with 1D blurs: 0.143 seconds.

5.3.2.8 Conclusion

One can see that there are many solutions to implement one algorithm on graphic cards. If the algorithm is separable, then the last solution is always the best one. If the algorithm is a convergent algorithm (the result does not change after n iterations) then running the kernels with different offsets is a good solution. If the algorithm requires only the direct neighbors at each iterations, then a good solution is to loop outside the kernel and to initialize the local memory as in listing 5.8.

5.4 Other methods

I also worked on algorithms to snap one object (source object) to another (target object). This step can replace the whole optical flow computation or can be applied after the computation for more precise matching.

5.4.1 Closest pixel research

One way to snap an object to the other is to convert the target object into a 3D image, and then, for each scan point of the source object, look for the closest non-zero voxel. It is possible to create a second 3D image of dynamically resizable arrays where each voxel contains a list of corresponding scan points. Hence finding the closest voxel means finding the n closest scan points. It is then easy to iterate through the n scan points to find the closest one. Finding the closest voxel can be done in four different ways based on a window around the considered voxel (scan point in image space).

Naive method The first method is to iterate through each voxel of the window to find the minimum Euclidian distance between the scan point (in image space) and the voxels of the window. This method is simple but requires many distance computations (as many as the number of voxels in the window), thus it is a really slow process.

A better idea is to order the search path so that the voxels the closest to the scan point (in image space) are checked first. There are three ways to make such a path. The first is to find a mathematic formula which allows this search path, but it would require complicated computations, especially in 3D, and in the end it could be slower than the first technic.

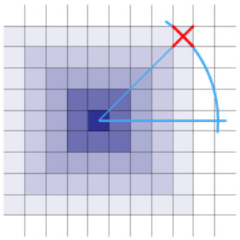

Figure 5.13: Outer shell search. Each shade of color represents one shell, we start with dark shells. The red cross is the first non-zero value; we must continue the search for one more shell.

23	20	12	16	24		
15	8	1	5	17		
11	4	0	2	9		
19	7	3	6	13		
22	14	10	18	21		

Figure 5.14: Look-Up Table of a circular search path.

Spherical search path: approximation by iterative outer shell search A second method is based on an approximation of the search path. By iteratively check the outer shell of the current region, we go through the closest voxels first. Yet, we cannot stop as soon as the first non-zero voxel is detected. In fact, one voxel on the next outer shell can be closer than the detected voxel. The research has to continue until the next shell further than the previous computed distance, as described on figure 5.13.

Spherical search path: Look-Up Table (LUT) The last method uses a loop up table as described on figure 5.14, a table in which the voxel order (the search path) is predefined. It is very simple to create this look up table just by iterating through each voxel of a window centred in the origin and by adding the distance and the voxel coordinates into a multimap. The multimap data structures orders the entries by keys (the distances) and allow multiple similar values (the voxels coordinates). Note that a 50 voxels wide window contains 125000 voxels (in 3D), that is 375000 voxel coordinates to store. This solution is really simple and fast for most windows (under 25 voxel wide windows).

Results I implemented the three algorithm and computed some research with a 10 voxel wide 3D window. The naive solution needs obviously 1000 iterations to find the closest non-zero voxel, whearas the Outer Shel method requires on average 58 iterations and the Look-Up table only 29.

5.4.2 Finding the closest scan point on GPU

It is not possible to properly implement a 3D image of resizable arrays on OpenCL. Without such image, the closest voxel research algorithms lead to a strong approximation since each scan point will be snapped into the discrete image space, and no linear interpolation is possible.

However, since handling huge arrays in parallel is really fast, it is possible for each scan point on the first object to iterate through each scan point of the other object. This can be implemented with a simple for loop on the kernel, but it is a typical parallel reduction problem. Parallel reduction means reducing an array of values to a single value in parallel (sum of all values, maximum of all values, etc.). The figure 5.15 illustrates parallel reduction.

It is also possible to use other type of data structures like a kd-tree to sort vertices and thus match scan points of both objects. The paper [19], [7] and [10] propose a way to implement a kd tree on GPU.

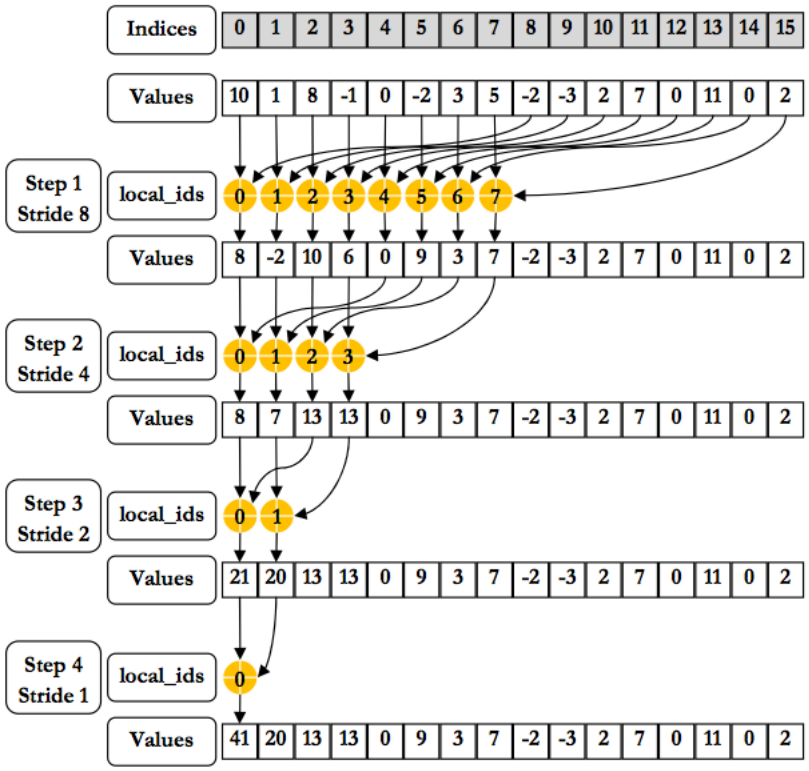

Figure 5.15: Parallel reduction without shared memory bank conflicts. Here the algorithm sums up every element of the array.

5.4.3 Gradient descent

One extremely efficient way to snap one object to another can be described as follow:

- Convert the target object into a 3D image,
- Blur the 3D image,
- For each scan point of the source object:

  Compute the gradient,

  Move toward the direction of the gradient until the maximum value is reached.

Since we know how the image was blurred, the value of each voxel indicates how far the voxel is from the closest edge. This means we can theoretically reach the maximum value in one step. We know we have to move of a distance of $(1 - currentValue) * blurRadius$ where $currentValue$ is the value of the voxel and $blurRadius$ is the radius of the blur. We just have to normalize the gradient and to multiply it by the computed distance to get the vector which leads to the closest scan point.

6 Conclusion

I must say that this internship was both interesting and challenging, thus very motivating.

Project management The project involved some complex and abstract theoretical background which was compelling and required some project management. I had to implement the algorithms step by step to validate each intermediate results. Some software management was also necessary to achieve the final product.

Working with OpenCL The most challenging part of the work was to define the best strategy to implement the algorithms correctly with OpenCL. The first task was to understand the OpenCL concepts. Then I could start the first programs and have a first overview of the classic bugs and encountered problems when working with OpenCL.

Debugging OpenCL programs require some experience since there is no OpenCL debugger for windows XP at this time. The only information available is the output of the OpenCL compiler, which is not always reliable. Hence, some program can contain important bugs but appear to work correctly. The same program can run perfectly with one memory setting and crash after rebooting the computer because the memory is not always correctly initialized.

In a general manner, the OpenCL is a brand new technology (first release on June 16, 2008) and online support is not developed and popular yet.

The project and VITRONIC I succeeded in implementing such a powerful algorithm, and I managed to apply it to an industrial problem. Furthermore, I designed a graphical user interface to really understand and adapt the algorithm to any problem the company could encounter for later use. To fully accomplish my work for the company, I was in charge of spreading the knowledge I acquired during this internship and my studies. Hence, I presented my work as well as an introduction to OpenCL and OpenGL to the developers and the project managers of VITRONIC. I was glad to work in the environment of VITRONIC, which had a great impact on my motivation.

Bibliography

[1] Weickert J. Alvarez L. and J. Sanchez. A scale-space approach to nonlocal optical flow calculations. *2nd International Conference on Scale-Space Theories in Computer Vision pages 235-246*, 1999.

[2] Aujol et al. Structure-texture image decomposition-modeling, algorithms, and parameter selection. *Int. J. Comp. Vis., 67(1):111-136.*, 2006.

[3] A. Chambolle. An algorithm for total variation minimizations and applications. *J. Math. Imaging Vis.*, 2004.

[4] A. Chambolle. Total variation minimization and a class of binary mrf models. *In Energy Minimization Methods in Computer Vision and Pattern Recognition. Pages 136-152.*, 2005.

[5] T. Chan and S. Esedoglu. Aspects of total variation regularized l1 function approximation. *SIAM J. Appl. Math., 65(5):1817–1837*, 2004.

[6] Nicu D. Cornea et al. 3d object retrieval using many-to-many matching of curve skeletons.

[7] Mike Houston Daniel Reiter Horn, Jeremy Sugerman and Pat Hanrahan. Interactive k-d tree gpu raytracing. 2007.

[8] Berthold K.P. Horn and Brian G. Rhunck. Determining optical flow. *Artificial Intelligence, pages 185-203*, 1980.

[9] J. Huang and D. Mumford. Statistics of natural images and models. *IEEE Conf. Comp. Vis. Pattern Recognit., pages 541-547, Fort Collins, CO, USA*, 1999.

[10] Rui Wang Kun Zhou, Qiming Hou and Baining Guo. Real-time kd-tree construction on graphics hardware. 2008.

[11] Rudin L., Osher S., and Fatemi E. Nonlinear total variation based noise removal algorithms. *Physica D, 60:259-268.*, 1992.

[12] Tim K. Lee and Mark S. Drew. 3d object recognition by eigen-scale-space of contours.

[13] Xinju Li and Igor Guskov. 3d object recognition from range images using pyramid matching. 2007.

[14] A. Marquina and S. Osher. Explicit algorithms for a new time dependent model based on level set motion for nonlinear deblurring and noise removal. *SIAM J. Sci. Comput., 22:387-405*, 2000.

[15] D. Mumford and J. Shah. Optimal approximation by piecewise smooth functions and associated variational problems. *Comm. Pure Appl. Math., 42:577-685.*, 1989.

[16] M. Nikolova. A variational approach to remove outliers and impulse noise. *J. Math. Imaging Vis. 20(1-2):99-120.*, 2004.

[17] Robert Osada et al. Matching 3d models with shape distributions.

[18] Thomas Pock. Fast total variation for computer vision. *Graz University of Technology*, 2008.

[19] Hans-Peter Seidel Stefan Popov, Johannes Gunther and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. 2007.

[20] Wikipedia. Hsl and hsv. 2005.

A Appendix

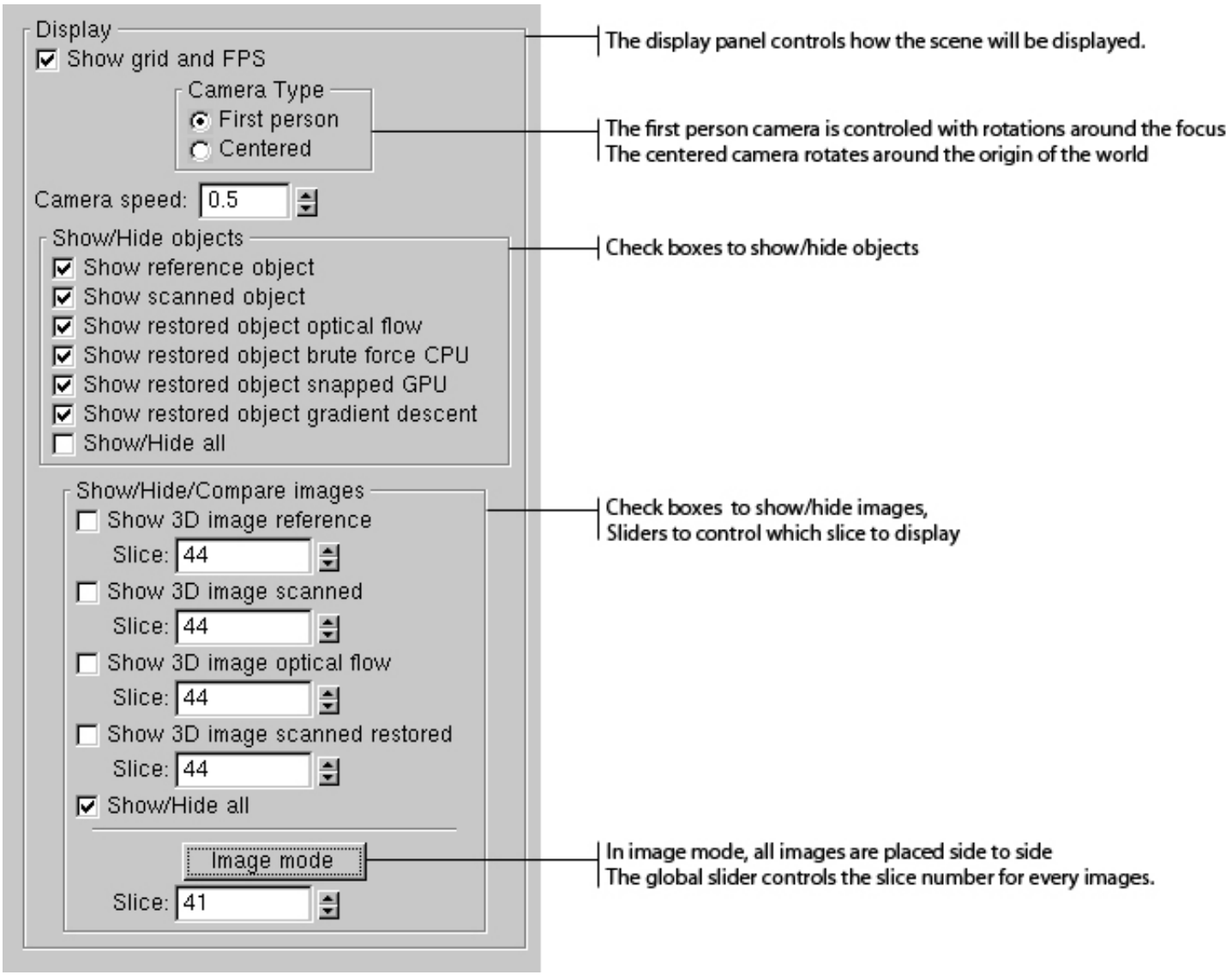

Figure A.1: User interface: 1st column

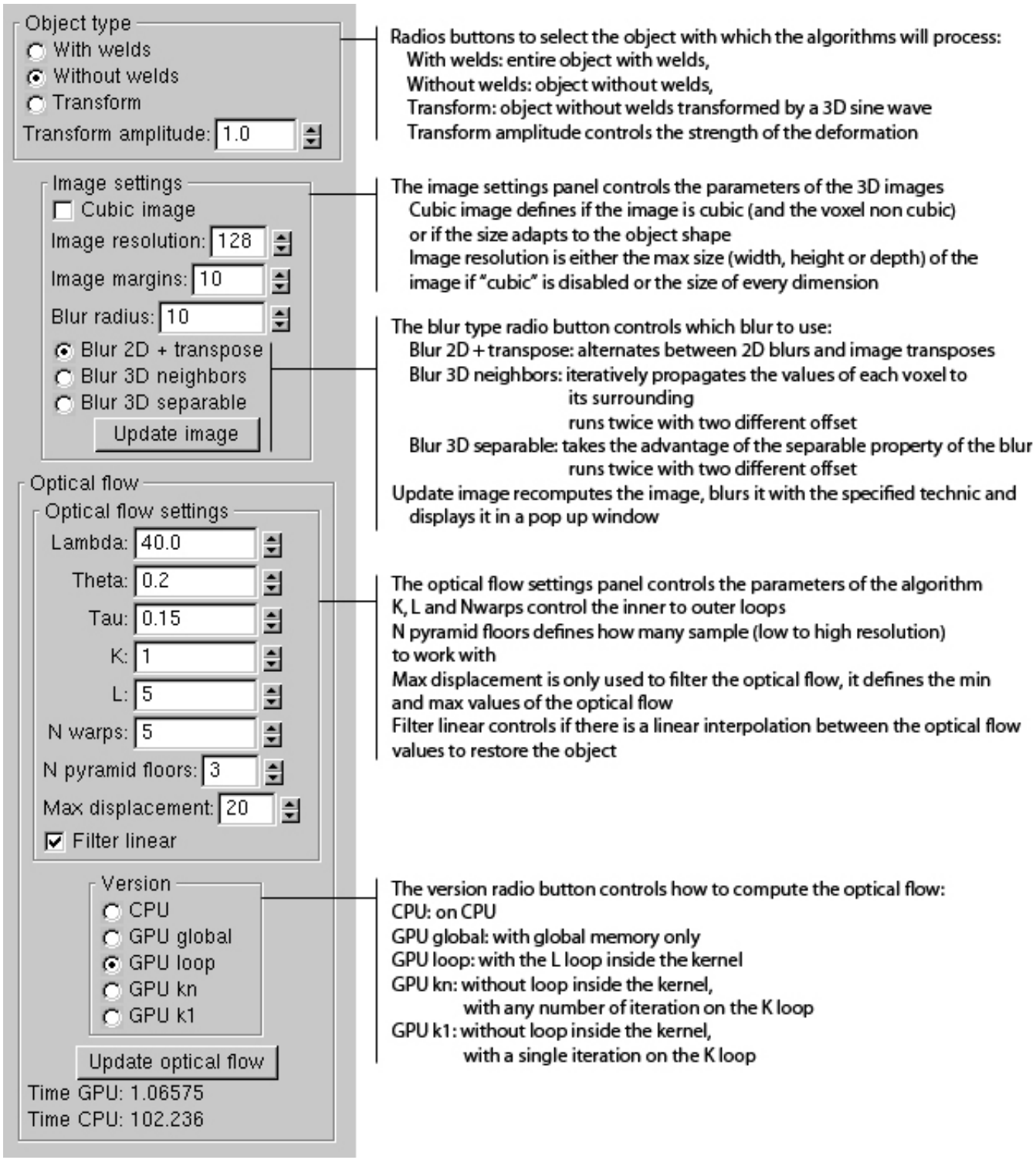

Figure A.2: User interface: 2nd column

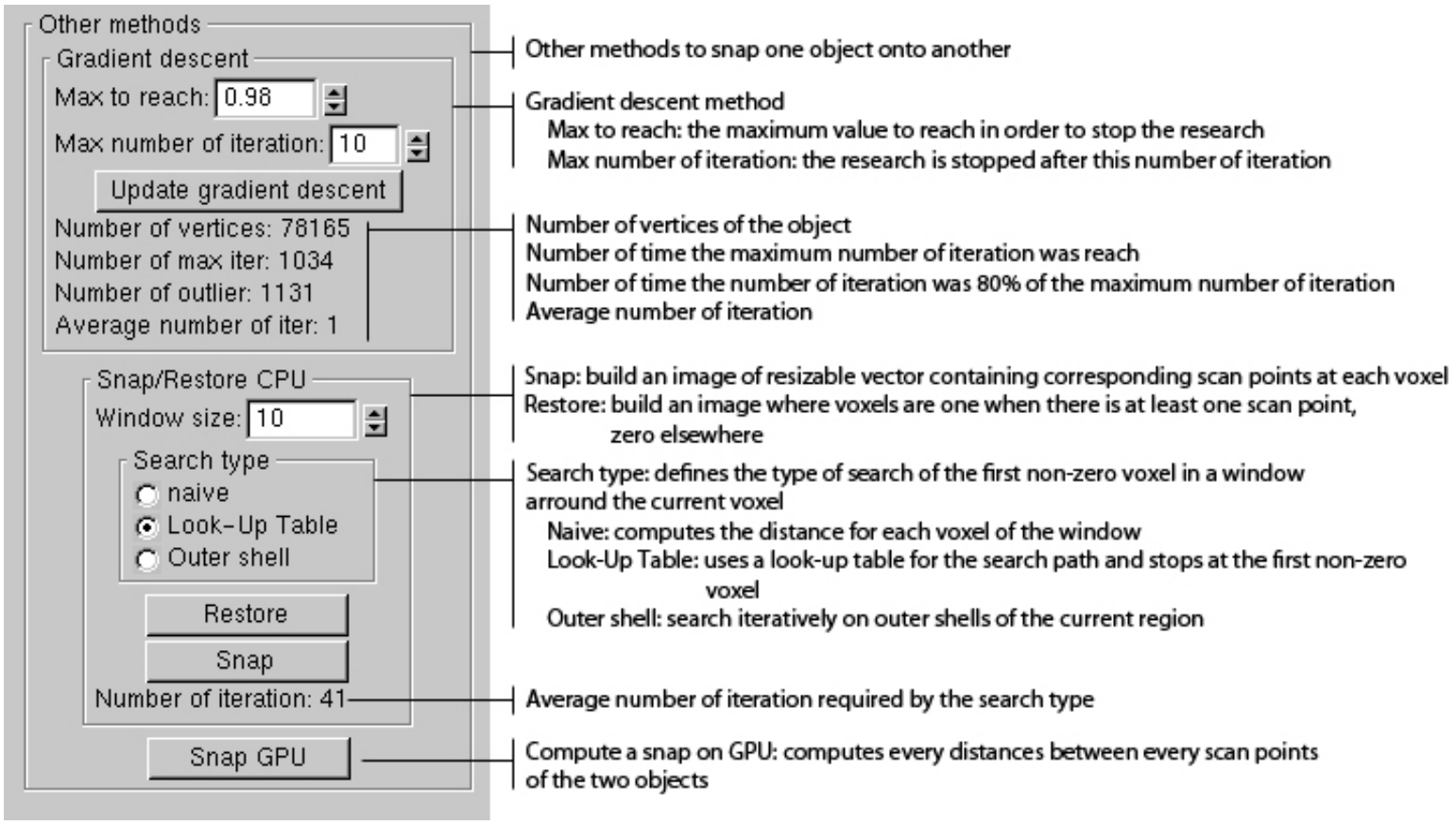

Figure A.3: User interface: 3rd column

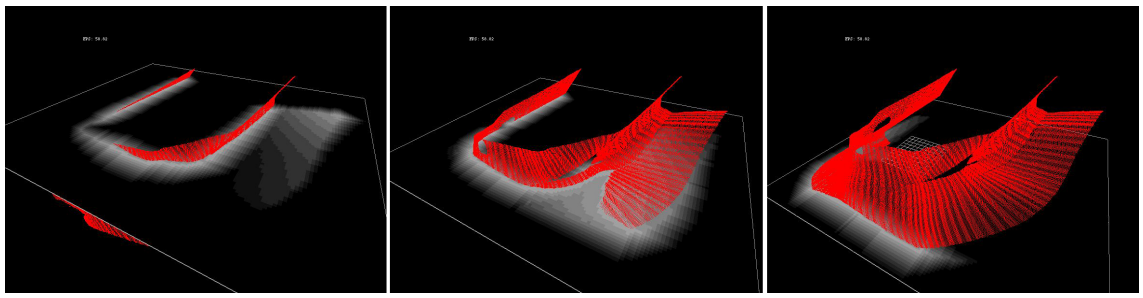

Figure A.4: User Interface: different slices of the 3D image projected on a quad.

Figure A.5: Noisy image (left) and restored image (right) with the ROF model on GPU. The GPU version can be hundreds of time faster than the CPU version when the number of iteration K is high.

Figure A.6: From left to right: original image, TVL1 denoising on CPU, TVL1 denoising on GPU

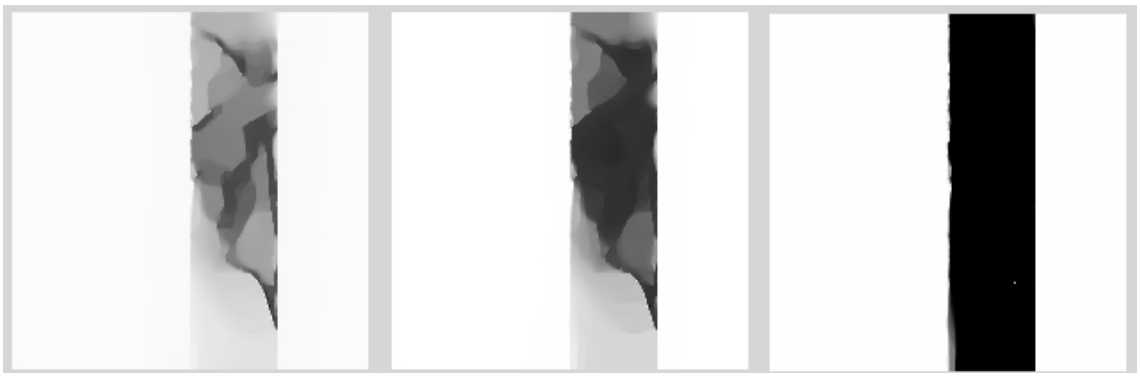

Figure A.7: Result of the 1D optical flow with lena.pgm and lena.pgm translated horizontally by two pixels on rows 100 to 150. From left to right: results after 3, 5 and 20 warps.

Figure A.8: 2D optical flow algorithm in matlab. From left to right: 1. two black shapes, 2. the same black shapes rotated, 3. the resulting flow field, and 4. the restored shapes.

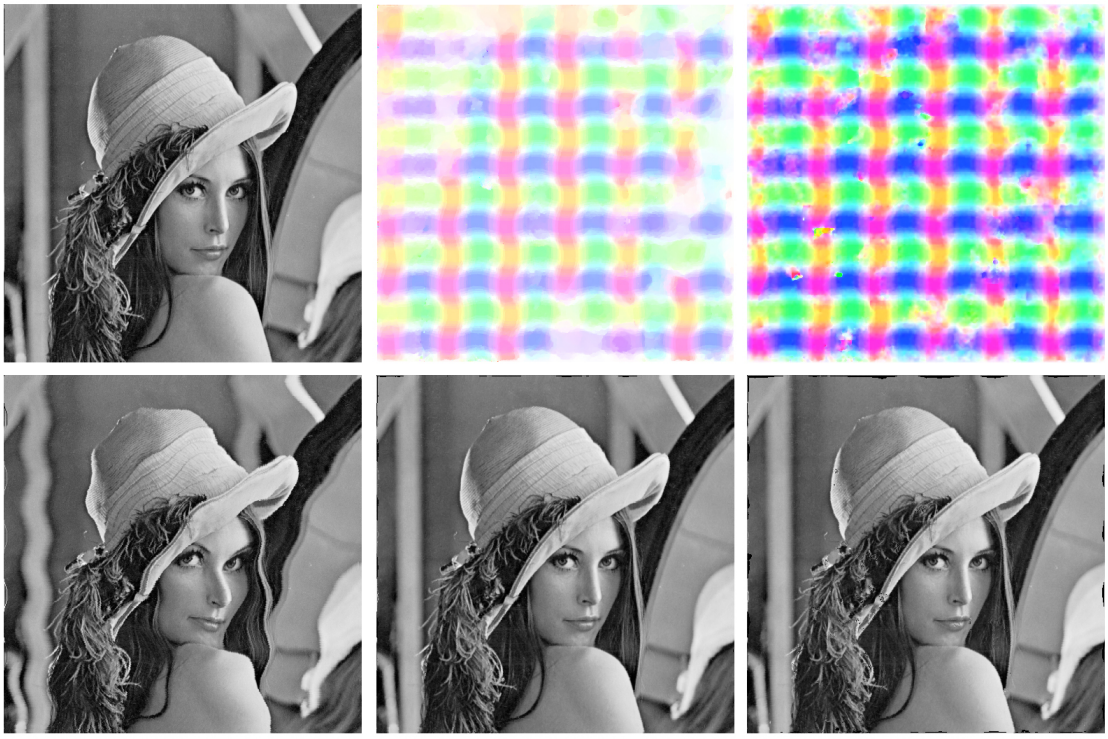

Figure A.9: Result of the 2D optical flow on lena.pgm and lena.pgm transformed by a 2D sine wave.

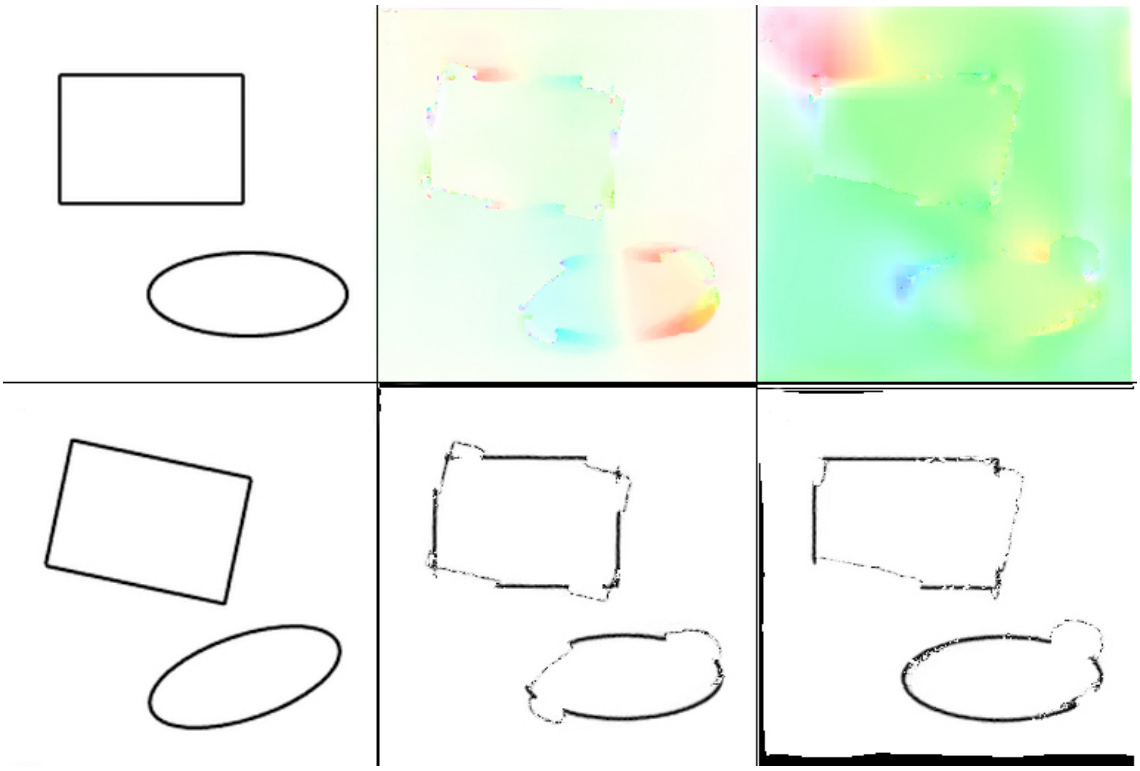

Figure A.10: Result of the 2D optical flow with two empty shapes.

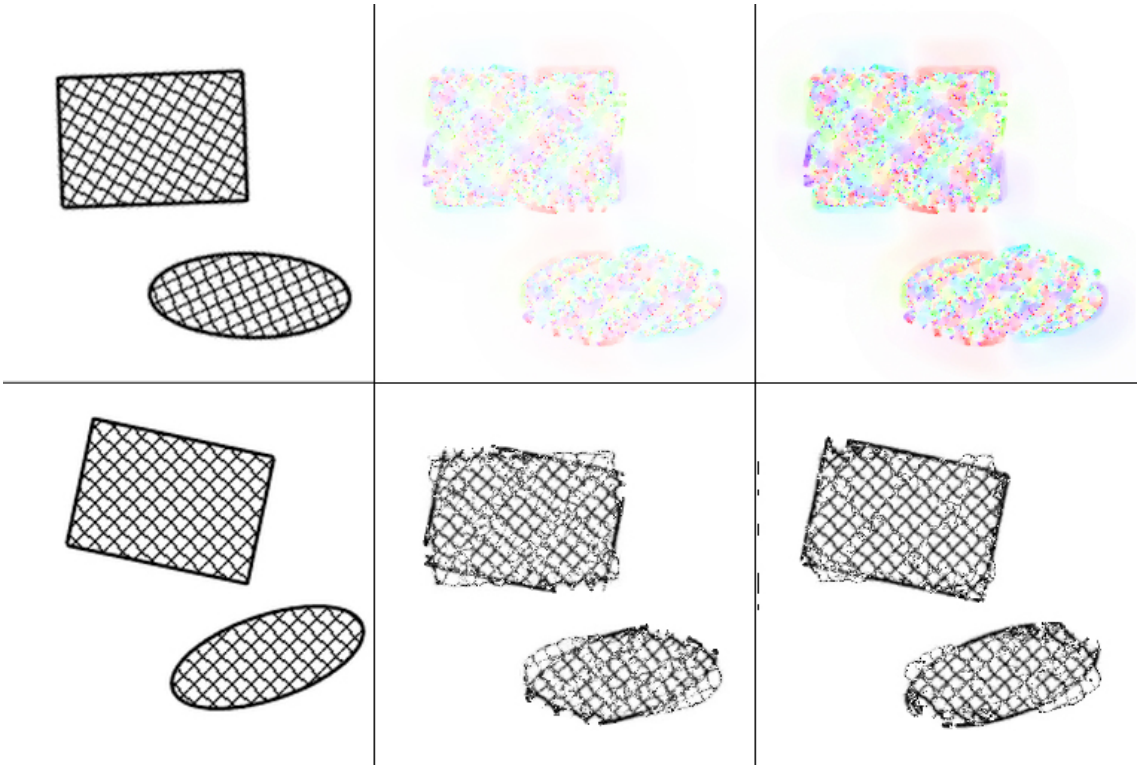

Figure A.11: Result of the 2D optical flow with two textured shapes.

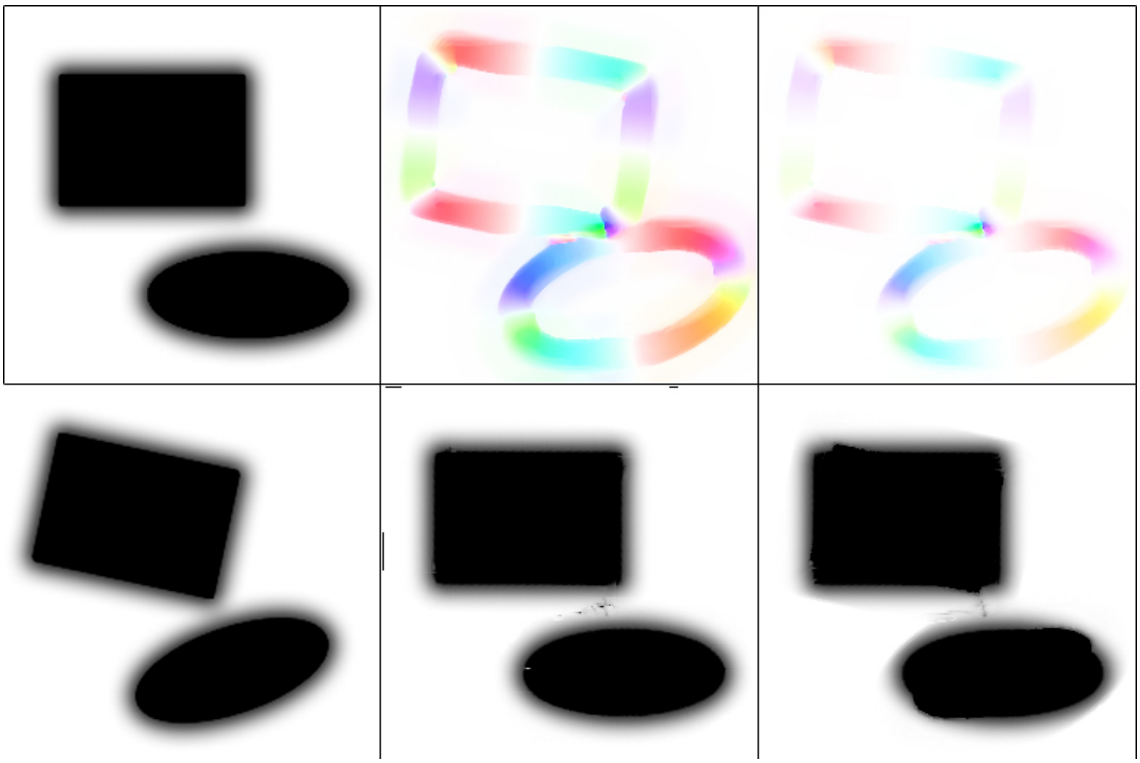

Figure A.12: Result of the 2D optical flow with two blurred shapes.

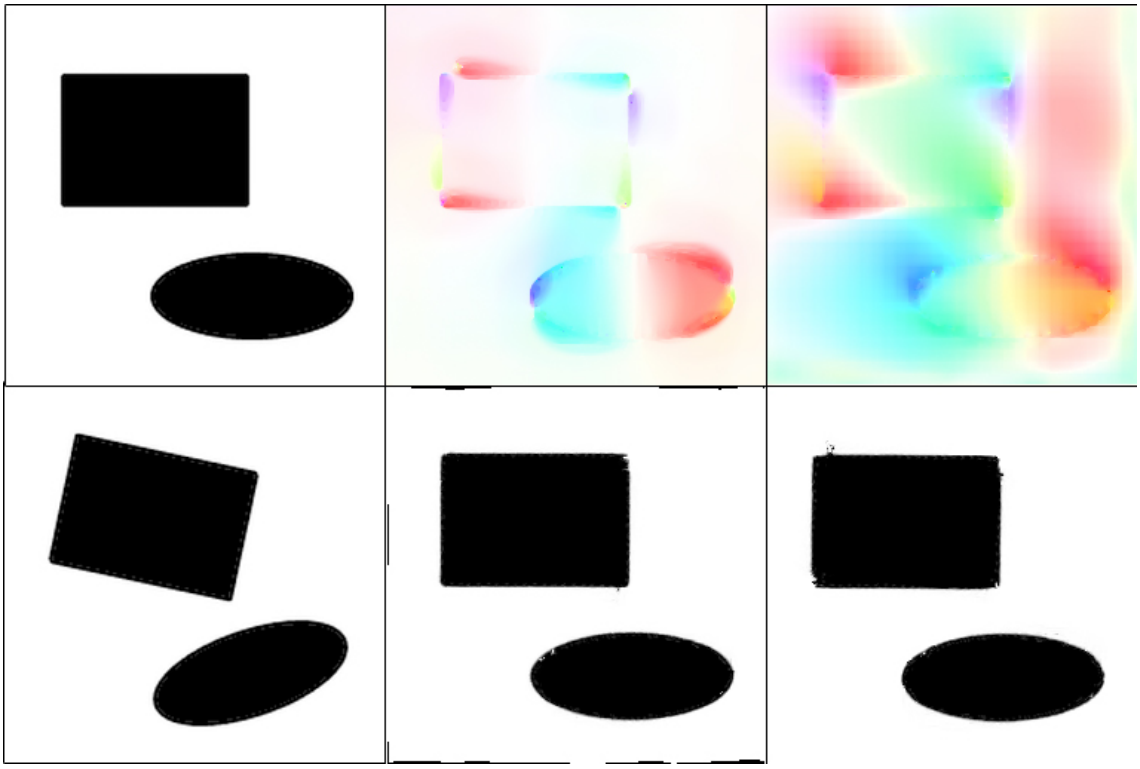

Figure A.13: Result of the 2D optical flow with two black shapes.

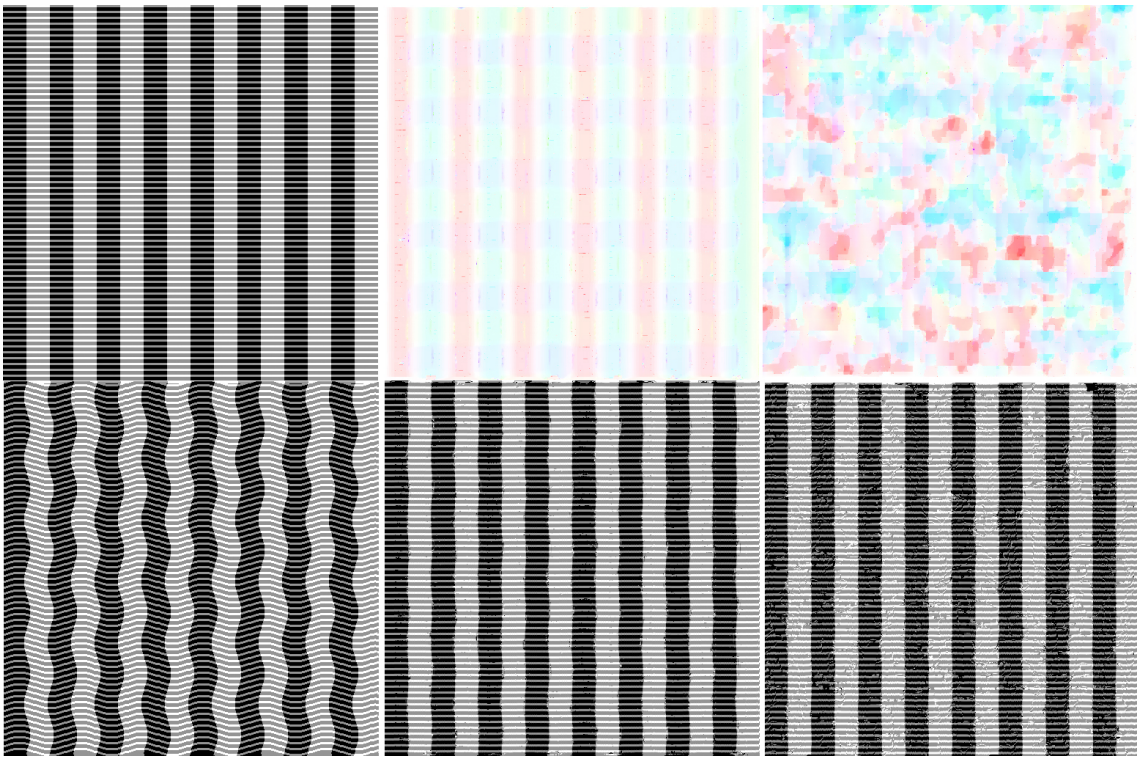

Figure A.14: Result of the 2D optical flow on a checker.

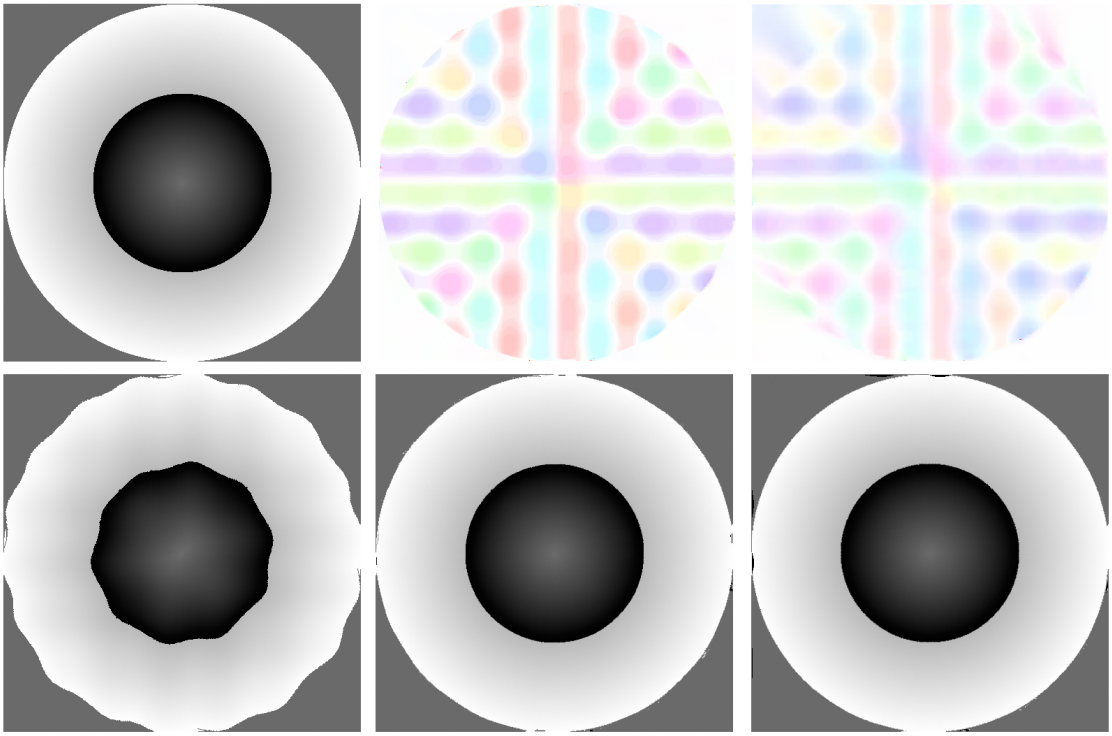

Figure A.15: Result of the 2D optical flow on a circular gradient.

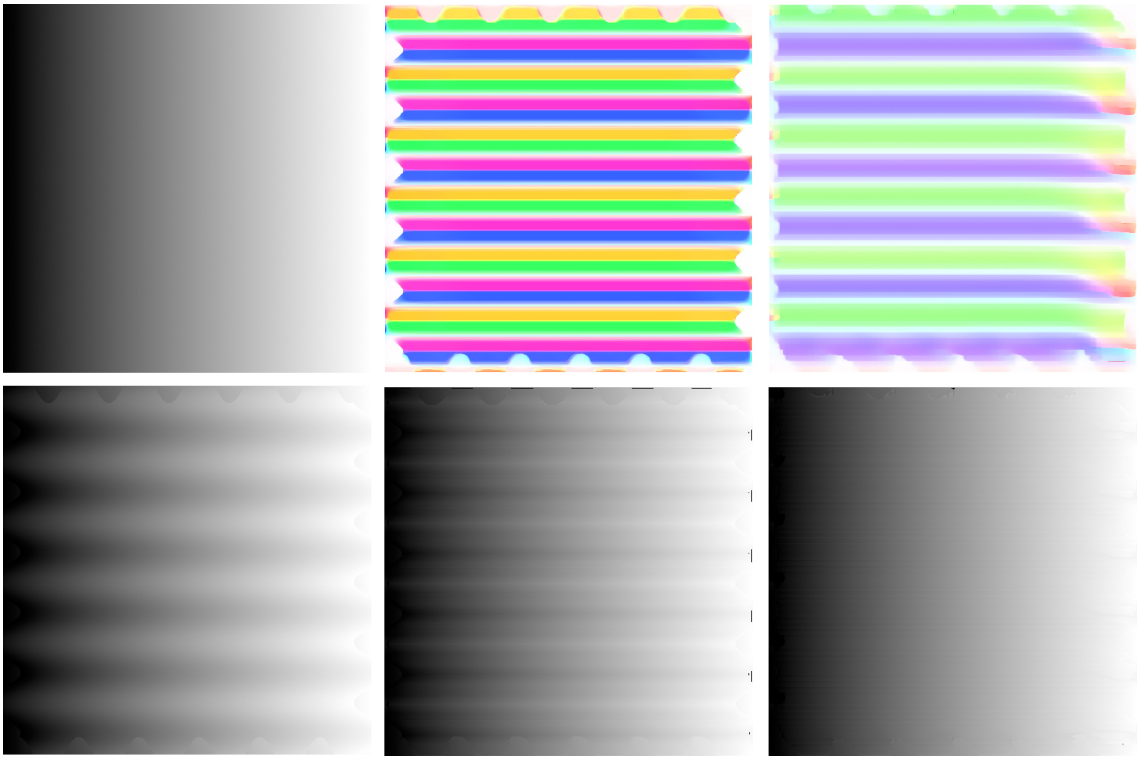

Figure A.16: Result of the 2D optical flow on a gradient.

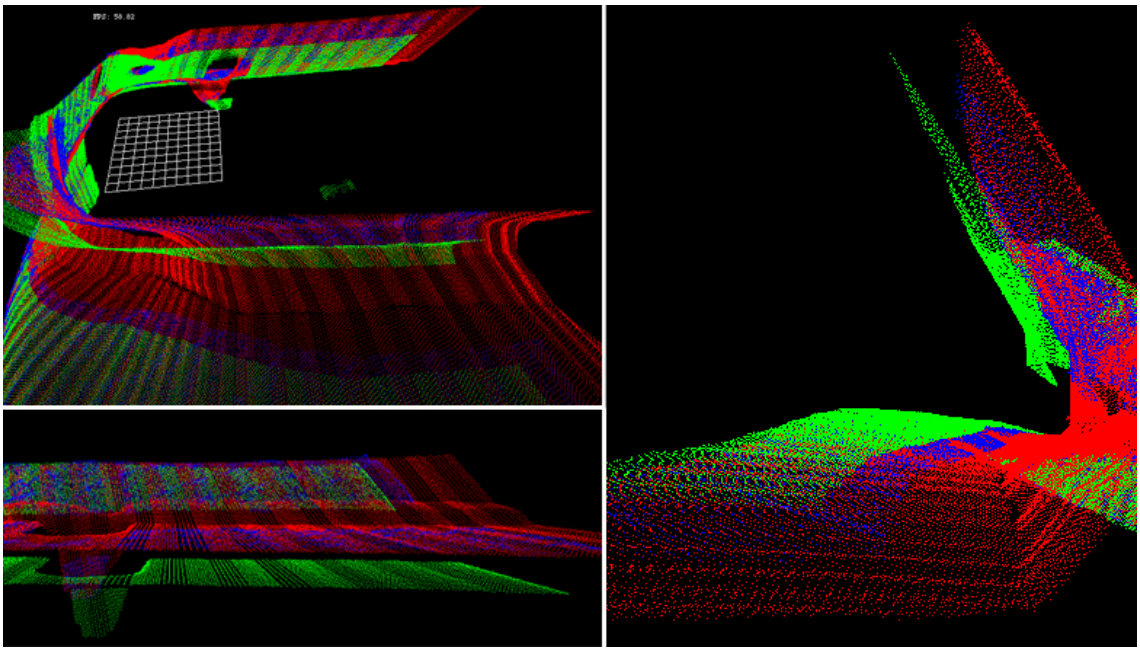

Figure A.17: Result of the 3D optical flow. The red object is the target object; the green object is the source object. The result is in blue; it fits correctly to the red object.