Integrated Computer-Aided Engineering 18 (2011) 391–400
DOI 10.3233/ICA-2011-0384
IOS Press

A low-cost 3D human interface device using GPU-based optical flow algorithms

Rafael del Riego[a], José Otero[b] and José Ranilla[b,*]
[a]*CRAZYBITS STUDIOS, Avenida Argentina 132, Gijón, Spain*
[b]*Department of Computer Science, University of Oviedo, Campus de Viesques S/N, Gijón, Spain*

Abstract. Except for a few cases, nowadays it is very common to find a camera embedded in a consumer grade laptop, notebook, mobile internet device (MID), mobile phone or handheld game console. Some of them also have a Graphic Processing Unit (GPU) to handle 3D graphics and other related tasks. This trend will probably continue in the next future and the pair camera+GPU will be more and more frequent in the market. Because of this, the proposal of this work is to use these resources in order to build a low-cost software-based 3D Human Interface Device (3D HID) able to run in this kind of devices, in real time without degrading the overall performance. This is achieved implementing a parallel version of an existing Optical Flow Algorithm that runs fully in the GPU without using it at full power. In this way, usual graphic processes coexist with Optical Flow computations. To the best of author's knowledge, this approach (a software-based 3D HID that runs fully in a GPU) is not found in academic research nor in commercial products prototypes. Indeed, this is the salient contribution of this paper. The performance of the proposal is good enough to achieve real time in low grade computers.

Keywords: Virtual 3D-HID, optical flow algorithms, GPGPU

1. Introduction

Personal computers, laptops, notebooks or smartphones have an integrated low-resolution camera (a back high-resolution camera in the case of smartphones and in some other cases a frontal low resolution one). Besides, most of them have a graphic processing unit (GPU), a specialized processor that offloads 3D/2D graphics rendering from the microprocessor. Consequently, a new paradigm that uses the GPU has arisen [28]. This concept turns the massive floating-point computational power of modern graphics accelerators into a general-purpose computing power. In certain applications, this allows us to increase the performance in some orders of magnitude compared to a conventional CPU [26,28].

Being aware of these capabilities developers try to use these resources to build new Human Interface Devices (HIDs). This feature is also a trend in the gaming industry. Indeed, it is a subset of "motion gaming" concept that includes Nintendo Wii,[1] Sony PlayStation Move and Eye/Eye Toy[2] or Microsoft Kinect,[3] dedicated platforms that aim to produce seamless gaming experience [6].

Other approaches include accelerometers available in the gaming device itself and/or adding a video projector in order to use any available planar surface as screen. This is the case of Moject (Mo-tion and pro-ject portmanteau) by Innovation in Mobile Technologies,[4] which attaches a pico-projector to an iPhone, or Microvision pico-projector video game gun-controller.[5]

A common approach is using a multi-touch surface in order to provide a suitable sensor to give a real feeling of interaction [25].

There are also software solutions that add motion sensitivity using the on board camera, with no extra

*Corresponding author: Dr. J. Ranilla, Department of Computer Science, University of Oviedo, Campus de Viesques S/N, Gijón, Spain, E33204. E-mail: ranilla@uniovi.es.

[1] http://wii.com/.
[2] http://playstation.com
[3] http://www.xbox.com.
[4] http://moject.com.
[5] http://www.microvision.com.

ISSN 1069-2509/11/$27.50

hardware needed but degrading the performance of the device, loading the processor with extra computations like, for example, the proposal of Engine Software for Nintendo Dsi.[6]

All this examples fall in the 3D Human Interface Device (3D-HID) category and in the Vision-based motion estimation with mobile devices [10]. Speaking in general, Computer Vision based HIDs proposals are common from gesture detection [5,7,20] to eye tracking algorithms [19]. Surveillance systems [13] bear some relationship with the techniques in this paper because gestures or posture have sometimes to be detected but cannot be considered HID.

In this work, a virtual 3D-HID similar to the previously commented approach for Nintendo DSi is proposed but with a distinctive feature (no extra hardware is needed): it uses the GPU to unload the CPU with motion-estimation computations and avoiding accelerometers, gyroscopes, and special cameras. To the best of author's knowledge, in the context of 3D-HID, there is no other work to date that uses the GPU of a device to estimate the motion of the device itself without diminishing the graphic performance of the system and being completely unobtrusive to the user.

Apart from the GPU usage, the proposed approach can be considered an estimation of egomotion [2] (depth from motion dual problem [16]), more specifically, a qualitative estimation of camera motion from Optical Flow (OF) [23] using a parallel implementation [8,9] in real time but without specialized hardware [15]. Real time is the most important goal because it has direct impact in usability [14] other than 30 frames per second performance would add an unacceptable delay to user experience.

In order to point out the aim of this paper, its main aspects are going to be reviewed. Thus, in Section 2 explains the optical flow algorithms. Section 3 is devoted to the built system. The experimental results are showed in Section 4 and finally, Section 5 summarizes the conclusions.

2. Optical flow

Optical Flow (OF) is the 2D vector field projection of the 3D velocities of object points. In Fig. 1 a pair of frames of a classic test sequence is shown with the true optical flow over imposed. As can be seen, the motion of the objects in the scene is well represented by the optical flow.

[6] http://www.engine-software.com/.

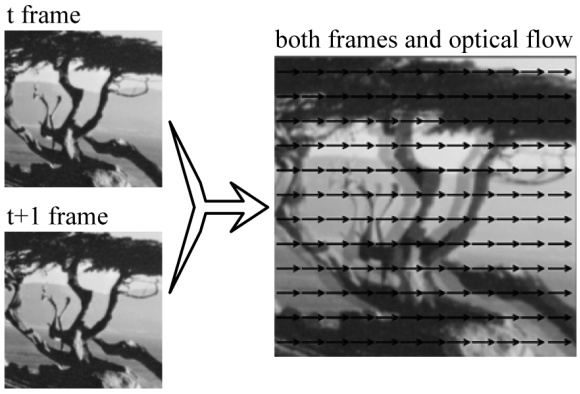

Fig. 1. OF example, from two frames (left) the motion of the scene is measured for each pixel. The resulting vector field (black arrows) is shown in the right side of the figure. Both frames are overimposed.

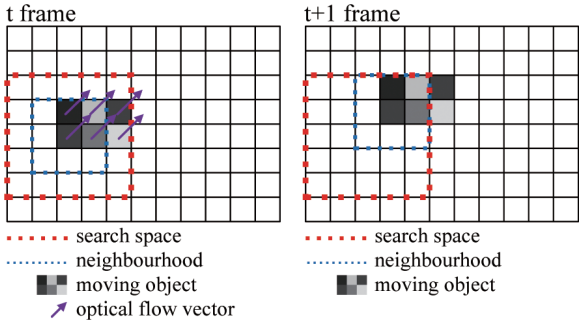

Fig. 2. OF computation using a block-matching algorithm. The neighborhood in the first frame (blue dotted square) is found in the second frame in different position. This displacement defines the OF vector for each pixel.

In the literature, OF algorithms are classified as follows: in correlation based techniques, in frequency based techniques and gradient based techniques.

Correlation based techniques or block matching algorithms (BMA) [1] try to maximize a measure of similarity between patches (taken from two consecutive frames) centered in a given pixel. The displacement that maximizes the selected measure divided by the time interval within the acquisition of the frames is the velocity of the pixel (see Fig. 2).

Frequency based techniques use a set of tuned spatiotemporal filters to search for the velocity of a pixel [11].

Gradient based techniques use the well known Optical Flow Constraint (OFC) shown in Eq. (1) in order to compute the OF [4]. In that equation, f is the gray level of a pixel, u and v are the components of that pixel OF in x and y axis, respectively.

$$-\frac{\partial f}{\partial t} = \frac{\partial f}{dx}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} = \frac{\partial f}{\partial x}u + \frac{\partial f}{\partial y}v \qquad (1)$$

Equation (1) makes the assumption that intensity changes (the spatiotemporal derivatives) in a sequence of images are only due to the movement of the objects in the scene: a single pixel will have constant brightness in the different positions that it takes during the sequence. Nonetheless, the Aperture Problem [17] states that there is no way to recover the complete OF vector using only local (one pixel) information, because Eq. (1) has two unknowns.

Some authors try to solve the aperture problem with the incorporation of some kind of global information involving a process of regularization [4].

Others perform a clustering of the OFCs themselves in order to find the most reliable one. Once obtained this, the corresponding normal flow to that OFC is achieved [18].

Another alternative is to analyze the measurements in the space of the velocities, that is, performing an estimation of the velocity with the results of many systems of OFC equations. Each system of equations is obtained from at least one pair of pixels in order to estimate the velocity. In this way, the analysis is performed directly in the domain of the data to be recovered, that is, the u, v space. Examples of this alternative are Otero et al. algorithm [21,22], Hierarchical Lucas-Kanade algorithm [6] or Schunck's proposal [24].

Recent approaches exist to improve OF algorithms results. For instance, in [27] procedures for the detection of areas where objects with different velocity overlap are proposed. Later, in this areas, the OF is improved using several diffusion filters.

Unfortunately, the OF accuracy improvement leads, obviously, to an increasing computational cost of the whole system. Recall that the aim of this paper is to achieve real time performance, because of this, approaches like this are discouraged.

In this paper, the approach in [24] was chosen as test bed, in order to ascertain if the execution speed up of an OF algorithm parallel implementation using a GPU is enough to achieve real time performance of the whole system. This approach has been chosen instead of the one in [21] because of the parallel version poor performance of this OF algorithm. This was found during the development of the system and it is not discussed here for simplicity's sake.

2.1. Hierarchical Lucas-Kanade algorithm

Hierarchical Lucas-Kanade (HLK) is composed of two procedures: an iterative one and a hierarchical one. The iterative part is very straightforward: arising

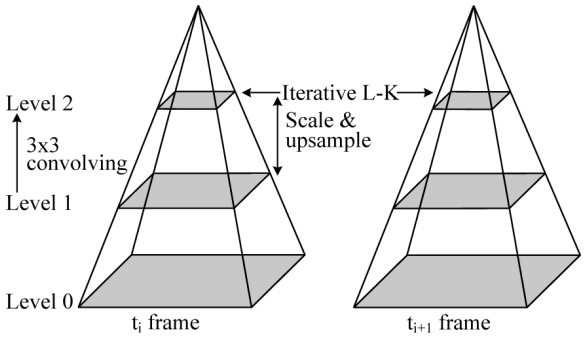

Fig. 3. How Hierarchical Lucas-Kanade algorithm works.

from OFC, the following equations Eq. (2) are obtained (see [6] for details):

$$G \doteq \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

$$\bar{b} \doteq \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I I_x \\ \delta I I_y \end{bmatrix} \tag{2}$$

$$\delta \bar{v}_{opt} = G^{-1} \bar{b}$$

where δI, I_x, I_y are spatiotemporal derivatives and w_x, w_y are integers that define the neighborhood size for the pixel at (p_x, p_y).

The goal is to find a motion vector that minimizes the error function in the following equation Eq. (3).

$$\varepsilon(\bar{v}_{opt}) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (A(x,y) - B(x+v_x, y+v_y))^2 \tag{3}$$

where A and B are usually termed the "first" and "second" images and v_x, v_y are the components of $\bar{v}_{opt}$.

The previous estimation Eq. (2) fails when the motion of the pixels is large. In order to refine the motion estimation, an iterative procedure is used: each movement estimation $\bar{v}_{opt}$ is used to warp the first frame making it closer to the second frame and an estimation of the *residual* of the motion vector is computed using again the previous equations with the transformed image as input. The procedure ends when the residual falls below a threshold (warped first frame very close to second frame) or the number of iterations is reached.

The hierarchical procedure uses what is called a "pyramid of images" (see Fig. 3): a set of different sized versions of an image. In particular, every "level" of the pyramid is an image constructed by halving the dimensions of the image in the level below starting

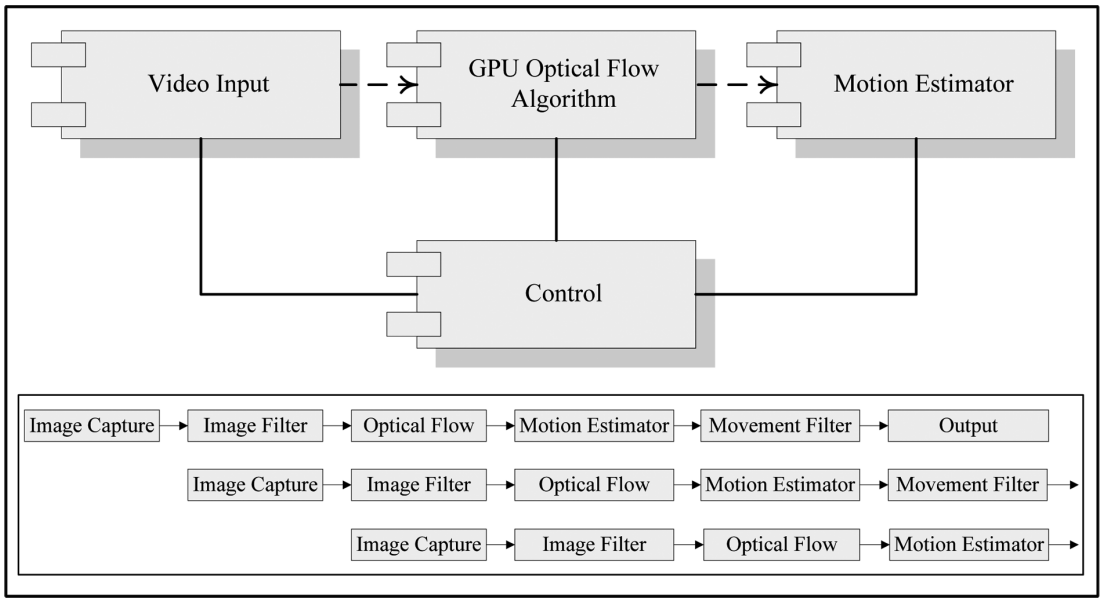

Fig. 4. Architecture of the proposed system.

from the original image in the first level. As the pixels of the image need to be sampled, smoothing of the input images is needed. The number of pixels at each level is four times lesser. The value of each pixel in a level is calculated convolving the level below using a 3 *by* 3 Gaussian kernel. Effectively, this smooths the image removing noise from the input. Moreover, the separability of this filtering benefits the parallelization of the whole system.

The other consequence is that the measurement of a movement in a level will be half the one in the level below. That means that the movement value is halved in every level. However as the same window size (in Lucas-Kanade algorithm) is applied, the search space is virtually doubled and it is now possible to track greater movements. This is important for a 2D-HID, otherwise, the user should self impose a maximum speed for his/her movements.

One thing to keep in mind is that as the images are smaller in each level and some points "vanish", it is also needed to remap the points where the motion will be estimated, exactly the opposite of the pyramidal decomposition.

At each frame, the process starts at the summit of the pyramid, making rough approximations of the movements. To accomplish these, the iterative process is applied at each level. The result of each level (multiplied by two) serves as starting point for the iterative procedure at the next level, until the base of the pyramid is reached, i.e. the full input image, is reached. The process ends when the last iteration is done at the base level, resulting in the final estimation of the motion.

3. Proposed system

The system presented in this work was designed with flexibility in mind, that is, it could be used to solve the proposed problem or to any other task that could be separated into pieces that inter-operate chained together in a lineal work-flow. Of course, non-lineal tasks could be adapted to fit into this design, although it is not the goal of this work.

The philosophy is not new; it has its background related to computer architecture designs, as it mimics the segmentation technique, common nowadays in the CPUs. Every step of the work-flow can be separated into tasks that are fitted into a "Segmentation Unit". In CPUs, the processing of every instruction is divided into small parts and then executed in specific components of the CPU. Thus, when an instruction has ended using the "Instruction Fetch" component, the next instruction can use it, having two instructions processed at the same time.

According to this philosophy, the system architecture is divided in four main subsystems: Video Input, Optical Flow Algorithm, Motion Estimator and Control. Each subsystem comprises several basic computing units (see Fig. 4). The "Segmentation Units" of the system are threads executed in parallel on the CPU

making a complete analogy with the CPU architecture. Of course, flaws and benefits of this framework are similar to those of CPU segmentation. Thus, good load balancing among pipelines is desired. It is up to the developers to know how the load of their work is best balanced.

As seen on Fig. 4, the use of this framework creates a work-flow that transforms an input image (provided it is not the first one of the sequence as, at least, two images are required in order to estimate the OF) into a set of values which quantify the motion estimated.

Other images are processed at the same time. This can be achieved even with a one-core CPU because the proposal makes the most of the combination CPU plus GPU.

The system delivers the heavier calculus of the production chain to the GPU. Thus, it releases the CPU from other tasks such as retrieving images from the input (from camera or from file during off-line testing), doing other minor calculations or lighter tasks as delivering the final result to the output.

At this point, it is easy to see that the proposed framework has two levels of parallelism. On the one hand, several functional units are working concurrently (including CPU and GPU) based on the pipelined technique. On the other hand, the OF computation unit runs inside the GPU exploiting data parallelism.

Each of these tasks is linked to the next by means of a buffer that lies between two "Segmentation Units". A buffer, as usually, is a place where information is stored to be used later. They manage the way the system behaves. Given the nature of the work, a ring buffer that could stop (i.e. put to sleep the internal threads, saving CPU time) the "Segmentation Units" when it has no data from the input buffer (or when it's output buffer is full) is an adequate solution.

Every "Segmentation Unit" executes several processes, so several logic parts can be processed in the same thread preventing overload due to thread management. It saves time as makes the most of the multi-core CPU capabilities without forcing the developer to know the internal mechanisms of threading.

The "Segmentation Units", which are "Control Units" of the processes itself, are in turn managed by a meta-controller unit, the Flow Controller. The users of the developed framework should emphasize on using this controller and other that can be obtained through this: the "Data Container". It acts as storage for data shared among Segmentation Units and can be used to pass some bits of info to the process and get info back from the process.

The following subsections provide details on the way the OF algorithm is parallelized (3.1 and 3.2) and how the output of the OF algorithm is evaluated (3.3).

3.1. Optical flow and the GPU

One of the core aspects of this work is the use of a heavily parallelized architecture, as the modern GPUs are, to gain extra computational power without degrading the performance of the CPU. This hardware is common nowadays in every PC. Even recent mobile devices present powerful GPUs capable to deliver these extra GFLOPs.

As seen in Section 2, algorithms based on the OF are computationally expensive. For example, in the case of a BMA algorithm, for a $n \times m$ pixels image, a search space of $p \times q$ pixels and a neighborhood of $s \times t$ pixels, the number of floating point operations is approximately $n \times m \times p \times q \times s \times t$. Because of this, a parallel implementation of HLK algorithm that runs on the GPU has been developed.

One key concept is that, due to the nature of each algorithm, not all of them are well suited for running efficient on GPUs. Recall that in this paper HLK [6] algorithm was chosen as a test bed. The exhaustive parallel implementation of the algorithms in [3] is an overwhelming task beyond the scope of this paper.

Finally, bear in mind that the *numerical* results of the parallelized algorithms are identical to the conventional implementations. Therefore, there is no need to compare the new implementations and the classic ones found in [3] in terms of numerical error. The salient aim of this paper is to achieve real time performance not an improvement on the OF accuracy.

3.2. CUDA implementations details

An API extension to the C programming language named CUDA (Compute Unified Device Architecture) is used to encode the algorithms that will be run on the GPU. CUDA, released by NVIDIA,[7] allows specified functions from a ordinary C program to run on compatible GPU's stream processors. This makes C programs capable of taking advantage of a GPU's ability to operate over substantial volume of data in parallel while still making use of the CPU when appropriate.

At first, the input image is loaded onto a CUDA "texture" which offers the improvements to manipulate im-

[7] http://www.nvidia.com.

ages such as interpolate between pixels needed for the algorithm. The next step after having the images loaded is building current image structure (i.e. pyramidal structure in case of HLK). Except for the first one, the following images and their structures are already loaded into memory as they were computed in the previous frames. When needed, the images are split into blocks that are rescaled independently (i.e. to form the next pyramid level in HLK).

According to CUDA documentation and the observed behavior during the development of the proposed system, one of the drawbacks of using graphics accelerators to improve computing performance is data loading process into (and back from) the GPU's memory because its high cost. The decision was to load the images as soon as possible and deliver to the GPU all the image-related calculus, as the CUDA offers some improvements to manipulate images. It is possible, due to the system architecture, to overlap communications (copy images from/to PC's memory to/from GUP's memory) and computation (floating point calculus on the GPU), that is, the movement of data between memories is made in asynchronous way to reduce its impact.

The next step involves the point filtering method to remove some unnecessary calculus for the next steps. As the calculus involves "reducing" the data to those that gather the useful information, some threads of the GPU are put to sleep while others gather the partial results. Finally, only one thread gathers the final result. Note that is very important not to get into thread synchronization too often writing to the same variable, as it incurs into performance penalty.

The core and final step of the GPU part of the global process ends with the estimation of the OF. As explained before, when parallelizing the point filter, the threads will operate independently among several parts of the image computing partial results of the matrices needed. Then, only some threads gather some partial results before only one thread summarizes all and retrieves the final real result. This is done at first for the needed matrices of the method (see [6]).

If the window size for the calculus is smaller enough to not overlap between different points where the OF will be estimated another optimization for the process is obtained. This is because it is not necessary to compute the spatial derivatives for the full image but only on the pixels in the window of each point which will be much lesser. The system used 240 points in an equally-distributed grid for 640 $\times$ 480 pixel images, with window sizes of 7 $\times$ 7 and 9 $\times$ 9. Thus, giving 11760 or 19440 pixels respectively against 307200 needed for the entire image, which represent a great performance gain. Therefore, the process is separated into parts of the image that are processed independently by replicated components of the GPU.

As CUDA abstracts some of the multithreading mechanisms implementation from the developer, only how the OF is estimated for one point has to be defined. The rest of the points are processed in parallel using the same code and sharing resources. Algorithm 1 shows the computational part of the CUDA's kernel to filter the points, where several points (as many as cores the GPU has) are processed at the same time (each point is automatically identified by local, threadIdx.x, blockDim.x, blockDim.y and blockIndex variables).

Algorithm 1. CUDA kernel.

```
__global__ void filterPoints(parameters){
...
  // Compute the differences for the pixels
  for (i = x; i < x + ratio; ++i)
    eT[local] = fabsf(tex2D(tex, i, y) -
                tex2D(texPre, i, y));
  __syncthreads();
  // Reduce the sum: first reduce each row
  if (threadIdx.x == 0) {
    for (i = local+1; i < window.width; ++i)
      eT[local] += eT[i];
  }
  __syncthreads();
  // Reduce the columns, hence get the total
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    for (i = blockDim.x; i < blockDim.y;
                          i+=blockDim.x)
      eT[local] += eT[i];
    if (eT[local] < thresholdPoints)
      velocities[blockIndex] = CUDAPoint2D();
  }
  __syncthreads();
}
```

3.3. Motion estimators

Optical flow field estimation is used in order to measure/detect the motion in the image sequence acquired with the cameras. The goal is to use the translations in X, Y and Z along with rotation in Z (see Fig. 5) as input signals to the proposed virtual interface.

In order to discriminate the predominant motion in the scene, the following operators are used:

– X and Y translations are measured as the average OF in the image:

$$(X, Y)_T = \frac{\sum_{i=0}^{i=n} \sum_{j=0}^{j=m} (v_x, v_y)_{i,j}}{nm} \qquad (4)$$

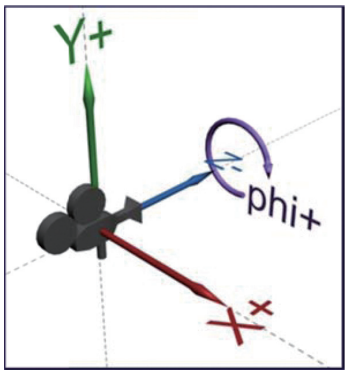

Fig. 5. Rotations and displacements to measure.

where $(X, Y)_T$ is the translation vector, $(v_x, v_y)_{i,j}$ is the OF vector for pixel (i, j), n, m are the number of rows and columns in the image.

– Z translation is measured with the divergence of the OF averaged across the image:

$$Z_T = \frac{\sum_{i=0}^{i=n} \sum_{j=0}^{j=m} \nabla \cdot (v_x, v_y)_{i,j}}{nm} \tag{5}$$

Equation (5) is evaluated and averaged for each OF value across the whole image.

– Z rotation is measured with the rotational of the OF averaged across the image:

$$Z_T = \frac{\sum_{i=0}^{i=n} \sum_{j=0}^{j=m} \nabla \times (v_x, v_y)_{i,j}}{nm} \tag{6}$$

The output of the OF algorithm is evaluated with the previous operators. The highest output defines the predominant motion in the scene.

All this calculations are done in a set of points uniformly distributed across the image for which their position is calculated by a simple internal algorithm. Given the desired number of points to accomplish the process, these points are always in the same position for the same image size. There is also another reason to proceed this way: the system does not focus on the movement of the elements of the scene but on the movement of the scene itself which is closer to the goal of the system.

Not all the points might be reliable to proceed. Points such as large uniform areas or points with minor changes between frames do not offer reliable information of the OF unless a too large window size is used. In order to prevent these points from been processed in a later stage of the process, a filter which removes them for the next step at each frame is proposed. It compares the value of the image intensity at each pixel and its neighborhood against a threshold Eq. (7), and then discards the points that do not fulfill the condition. What is more, the time inverted filtering is recovered as time saved in the heavier part of the calculus because some points are filtered before the main part of the process. This is summarized in Eq. (7), where $I(p_x, p_y)$ is image intensity at (p_x, p_y), I_{max} is the intensity maximum in the neighborhood and T is a threshold.

$$\sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} I(p_x, p_y) \leqslant \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} T \cdot I_{max}(p_x, p_y) \tag{7}$$

In order to remove pixels with small changes between frames, for every point and its corresponding neighborhood the difference in its intensity with the previous image is computed, added up and compared against a threshold made using the input value specified. Lowering it will have more points passing the filter.

Low illumination leads to noise in the image and to the false detection of small movements due to the noisy pixels that appear and disappear. It was decided to filter the OF vectors that are below a threshold in order to minimize that error.

Table 1
Equipments used to evaluate the performance

Desktop (EQ1)	CPU Intel Core2 Quad Q9550 2.83 GHz
	RAM 4 GB
	GPU nVidia GeForce 9500 GT
	Web cam 1: Logitech Quickcam E2500
Laptop (EQ2)	CPU AMD Athlon 64 3000+ (1.8 GHz)
	RAM 1024 MB
	GPU nVidia GeForce 9500M
	Web cam 1: Logitech Quickcam E2500
	Web cam 2: Logitech Webcam C200

4. Experiments

Several experiments were done in order to test de proposed system with the chosen OF algorithm. Also, two equipments are used to evaluate the performance of the system (see Table 1). Note that in both equipments, the maximum frame rate is restricted by the camera quality or by the usb-connection speed.

One kind of experiments was made in real time with the webcams provided by the equipments in order to test the functionality of the system as a software-based 3D Human Interface Device. First, the behavior of the system was tested when the element endowed with motion is the webcam (see Fig. 6). Second, some tests

Fig. 6. Several frames extracted from a test sequence moving a webcam in an indoor environment. Top: camera moving from bottom to top. Shown frames span over 0.48 seconds. Bottom: camera moving from left to right. Shown frames span over 0.32 seconds.

Fig. 7. Several frames extracted from a test sequence with a subject moving in front of a web cam. The webcam is placed at right bottom corner. *Exactly* the same software is running. Only left to right movement is shown, vertical movements are detected but they are too small (as the jumps of the subject are) to be noticed printed. See the online video for details.

were made moving the hand/body of a person in front of the webcam (see Fig. 7). An URL[8] with small videos of this kind of experiment is provided, because (obviously) this kind of information is not well suited for a printed and static media. During the experiments it was found that translations are well detected.

In order to quantify the performance of the proposed system, and test if basic movements are detected correctly, making quantitative measurements, some additional tests were made. Thus, some synthetic videos of scenes were generated with a well defined, easy to perceive movement in an axis: X, Y, Z and the rotation along Z axis. In each video only one kind of movement was present. To make it more complete 50 frames per video are provided. The motion has constant velocity to avoid errors or noise. This procedure allows customizing the videos greatly adjusting the movement to the one to be tested.

In Fig. 8 some frames of synthetic videos are showed. Each one corresponds to a specific movement.

The system detects easily the motion in different axis. Some errors may appear only when velocity module falls below a threshold. This could happen with real images mainly due to image acquisition and illumination issues (flickering or low illumination).

[8] http://di002.edv.uniovi.es/~jotero/Video_3DInterface.avi.

During the experiments, it was found that X and Y translations are easily detected and measured. Z translation is only detected because small displacements in X or Y directions (by the user) lead to relatively high values compared with the divergence of the OF field. To avoid this, before divergence estimation using Eq. (5), the OF field is scaled by a suitable constant. For the other motion detectors, the OF field is kept unchanged. The amount of motion cannot be accurately measured but if a threshold is used it can simulate a click.

Rotation in Z axis is another kind of movement which is useful as input signal (click alike) in the virtual interface. System behavior is summarized in Table 2.

The performance in terms of number of frames per second processed was also tested along with computing power consumption. Using CUDA implementation the frame rate increases to 30 fps (the maximum frame rate of the camera or the usb-connection speed) in both systems. Moreover, standard GPU tasks are not degraded and the CPU can be fully dedicated to other tasks. Thus, the user experience is real-time alike. The results obtained for these experiments are shown in Table 3.

On the one hand, equipment EQ1 always gets 30 frames per second (fps) regardless of the usage of the GPU. However, the percentage of CPU usage is lower when the GPU is used as general-purpose computing device. On the other hand, EQ2 obtains 17 fps when

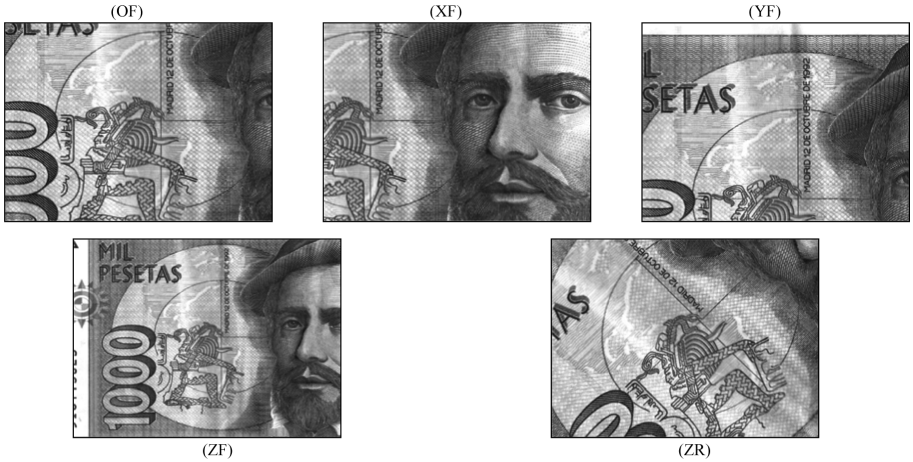

Fig. 8. Some frames of synthetic videos, each one corresponding to a specific movement. From top to bottom and left to right, original frame (OF), X (XF), Y (YF), and Z (ZF) translations and rotation around Z (ZR) are shown.

Table 2
System response summary

Movement	System Response
X	Measured / Useful
Y	Measured / Useful
Z	Detected / Useful
Z Rot	Detected / Useful

Table 3
Performance achieved by the system with/without using the GPU. fps: Frames per second

	EQ1	EQ2
fps without using the GPU	30	17
% usage of the CPU	medium	high
fps using the GPU as computing device	30	30
% usage of the CPU	low	low

the GPU is not used, and the performance of other tasks is degraded (high percentage of CPU usage).

5. Conclusions

In this work is shown how to build a Virtual 3D Human Interface Device by using standard resources of the current computers: the integrated camera (or Web cam) and the graphic processing unit (GPU). The system applies OF techniques and uses CUDA to exploit the capabilities of the GPU as stream processor. This approach keeps the CPU of the computer fully available to other tasks and their performance is not degraded. The presented solution is simple and clean since no extra hardware is needed and relies only on software.

A HLK algorithm parallel version was designed and implemented to build the system together with adding a filtering stage to the original algorithms that avoids processing points in large blank areas or with minor changes between frames.

Using the proposed approach, the frame rate and resolution is now limited only by the characteristics of the camera. Because of this, no delay is added and user experience is real-time. Alike, computer's performance is not being degraded without the need of powerful/additional hardware.

As stated before, some issues exist regarding with the Z axis. One possible solution to address this is following the approach in [23]. Probably this will lead to accuracy *vs.* parallel efficiency trade-off: perhaps the most accurate OF algorithm is not well suited for a parallel implementation.

Acknowledgements

This work is supported by the Spanish Office of Science and FEDER, under the grants TIN2007-61273, TIN2008-06681-C06-04 and TIN2010-14971.

References

[1] P. Anandan, A Computational Framework and an Algorithm for the Measurement of Visual Motion, *International Journal of Computer Vision* **2** (1989), 283–310.
[2] X. Armangué, H. Araújo and J. Salvi, A Review on Egomotion by Means of Differential Epipolar Geometry Applied to the Movement of a Mobile Robot, *Pattern Recognition* **36**(12) (2003), 2927–2944.
[3] J.L. Barron, D.J. Fleet and S.S. Beauchemin, Performance of Optical Flow Techniques, *International Journal of Computer Vision* **12**(1) (1994), 43–77.

[4] B.K.P. Horn and B.G. Schunck, 1992. Determining Optical Flow, in: *Shape Recovery*, L.B. Wolff, S.A. Shafer and G.E. Healey, eds, Jones and Bartlett Publishers, Inc., USA, pp. 389-407.

[5] A. Besinger, T. Sztynda, S. Lal, C. Duthoit, J. Agbinya, B. Jap, D. Eager and G. Dissanayake, Optical Flow Based Analyses to Detect Emotion from Human Facial Image Data, Expert Systems with Applications, Volume 37, Issue 12, December 2010, Pages 8897-8902, ISSN 0957-4174, DOI: 10.1016/j.eswa.2010.05.063.

[6] J.-Y. Bouguet, Pyramidal Implementation of the Lucas-Kanade Feature Tracker, Description of the Algorithm, Technical report, Intel Corporation, Research Labs, 1994.

[7] L. Chern-Sheng, H. Chien-Wa, C. Kai-Chieh, S.-S. Hung, H.-J. Shei and M.-S. Yeh, A Novel Device for Head Gesture Measurement System in Combination with Eye-controlled Human-machine Interface, *Optics and Lasers in Engineering* **44**(6) (2006), 597–614.

[8] J. Díaz, E. Rosa, R. Agísa and J. Luis Bernier, Super-pipelined High-performance optical-flow Computation Architecture, *Computer Vision and Image Understanding* **112**(3) (2008), 262–273.

[9] M. Fleury, A.F. Clark and A.C. Downton, Evaluating Optical-flow Algorithms on a Parallel Machine, *Image and Vision Computing* **19**(3) (2001), 131–143.

[10] J. Hannuksel, P. Sangia and J. Heikkilä, Vision-based Motion Estimation for Interaction with Mobile Devices, *Computer Vision and Image Understanding* **108**(1–2) (2007), 188–195.

[11] D.J. Heeger, Optical Flow using Spatiotemporal Filters, *International Journal of Computer Vision* **1**(4) (1988), 279–302.

[12] J. Loviscach, Playing with All Senses: Human–Computer Interface Devices for Games, *Advances in Computers* **77** (2009), 79–115.

[13] Y. Liu and C. Pomalaza-Ráez, 2010. On-chip Body Posture Detection for Medical Care Applications using Low-cost CMOS cameras, *Integrated Computer-Aided Engineering* **17**(1) (January 2010), 3–13.

[14] C. Manresa-Yee, P. Ponsa, J. Varona and F.J. Perales, User Experience to Improve the Usability of a Vision Term-based Interface, Interacting with Computers, *Now Available* **22**(6) (2010), 594–605.

[15] J.L. Martín, A. Zuloaga, C. Cuadrado, J. Lázaro and U. Bidarte, Hardware Implementation of Optical Flow Constraint Equation using FPGAs, *Computer Vision and Image Understanding* **98**(3) (2005), 462–490.

[16] O. Millnert, T. Goedemé, T. Tuytelaars, L. Van Gool, A. Hüntemann and M. Nuttin, Range Determination for Mobile Robots using an Omidirectional Camera, *Integrated Computer-Aided Engineering* **14**(1) (2007), 63–72.

[17] D.W. Murray and B.F. Buxton, Experiments in the Machine Interpretation of Visual Motion, MIT Press, Cambridge, USA, 1990.

[18] P. Nesi, A. del Bimbo and D.B. Tzvi, Robust Algorithm for Optical Flow Estimation, Journal on Computer Vision, *Graphics and Image Processing: Image Understanding* **61**(2) (1995), 59–68.

[19] B. Noureddin, P.D. Lawrence and C.F. Man, A Non-contact Device for Tracking Gaze in a Human Computer Interface, *Computer Vision and Image Understanding* **98**(1) (2005), 52–82.

[20] R.G. O'Hagan, A. Zelinsky and S. Rougeaux, Visual Gesture Interfaces for Virtual Environments, *Interacting with Computers* **14**(3) (2002), 231–250.

[21] J. Otero, A. Otero and L. Sánchez, Mode Based Hierarchical Optical Flow Estimation, *Machine Graphics & Vision* **10**(4) (2001), 489–501.

[22] J. Otero, A. Otero and L. Sánchez, 3D Motion Estimation of Bubbles of Gas in Fluid Glass, Using an Optical Flow Gradient Technique Extended to a Third Dimension, *Machine Vision and Applications* **14**(3) (2003), 185–191.

[23] P. Sang-Cheol, L. Hyoung-Suk and L. Seong-Whan, Qualitative Estimation of Camera Motion Parameters from the Linear Composition of Optical Flow, *Pattern Recognition* **37**(4) (2004), 767–779.

[24] B.G. Schunck, Image Flow Segmentation and Estimation by Constraint Line and Clustering, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**(10) (1989), 1010–1027.

[25] D. Valkov, F. Steinicke, G. Bruder and K.H. Hinrichs, 2010. A Multi-Touch Enabled Human-Transporter Metaphor for Virtual 3D Traveling. In Proceedings of the 2010 IEEE Symposium on 3D User Interfaces, 79–82, IEEE Press.

[26] V. Volkov and J.W. Demmel, 2008, Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08), 1–11, IEEE Press, Piscataway, NJ, USA.

[27] S. Viet-Uyen Ha and J.W. Jeon, Readjusting Unstable Regions to Improve the Quality of High Accuracy Optical Flow, *IEEE Transactions on Circuits and Systems for Video Technology* **20**(4) (2010), 540–547.

[28] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, GPU Computing, *Proceedings of the IEEE* **96**(5) (2008), 879–899.