

A generic library for structured real-time computations: GPU implementation applied to retinal and cortical vision processes

Jean-Charles Quinton

▶ **To cite this version:**

Jean-Charles Quinton. A generic library for structured real-time computations: GPU implementation applied to retinal and cortical vision processes. Machine Vision and Applications, 2010, 21 (4), pp.529-540. 10.1007/s00138-008-0180-9 . hal-01966569

HAL Id: hal-01966569
https://hal.science/hal-01966569

Submitted on 28 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Machine Vision and Applications manuscript No.
(will be inserted by the editor)

A generic library for structured real-time computations

GPU implementation applied to retinal and cortical vision processes

Jean-Charles Quinton[1]

Computer Science Research Institute, National Polytechnic Institute, University of Toulouse, 2 rue Charles Camichel BP 7122 - F 31071 Toulouse Cedex 7, France

Received: date / Revised version: date

Abstract Most graphics cards in standard personal computers are now equipped with several pixel pipelines running shader programs. Taking advantage of this technology by transferring parallel computations from the CPU side to the GPU side increases the overall computational power even in non graphical applications by freeing the main processor from an heavy work. A generic library is presented **to show how anyone can benefit from modern hardware by combining various techniques with little hardware specific programming skills. Its shader implementation is applied to retinal and cortical simulation.** The purpose of this sample application is not to provide a correct approximation of real center surround ganglion or middle temporal cells, but to illustrate how easily intertwined spatiotemporal filters can be applied on raw input pictures in real-time. Requirements and interconnection complexity really depend on the vision framework adopted, therefore various hypothesis that may benefit from such a library are introduced.

Send offprint requests to:

1 Introduction

Whether one tries to implement biological or artificial vision systems, low-level processes are massively parallel. Visual perception is considered active in frameworks such as interactivism, enaction or ecological vision. Such approaches introduce temporal patterns of local anticipations in the retinal field resulting from eye saccades, body movements or environment dynamics. To some extend, top-bottom modulations provide a way to reduce computations, concentrate on specific features and cope with environmental/perceptual noise. Yet unanticipated motion, rapid changes in brightness or any striking event

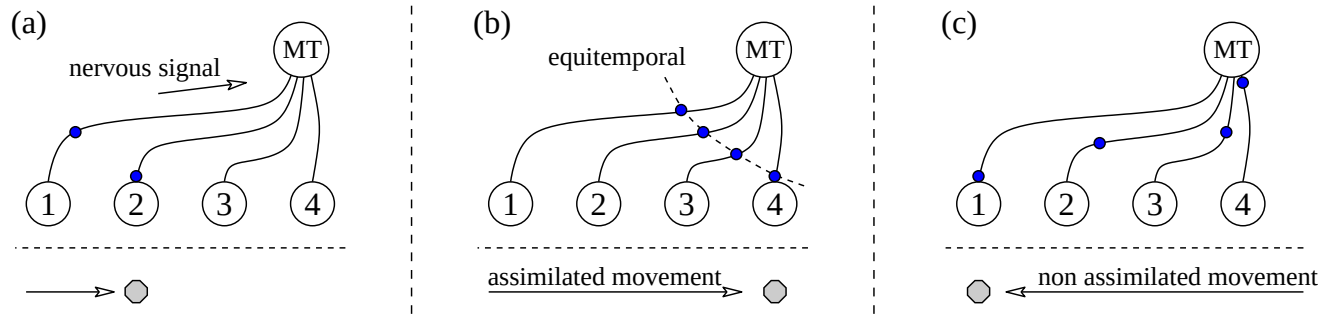


Fig. 1 Simplified one dimensional representation of anticipations performed by a Middle Temporal cortical cell (MT). (*a*) Inputs cells (1 and 2) get activated when an object moves in the field of view. The higher the input cell index (from 1 to 5), the lower the propagation time to the MT cell. (*b*) An assimilated movement will lead to the activation of the MT cell resulting from the synchrony of the signals. (*c*) Any movement that is either too fast, too slow or inverse will not lead to recognition.

relative to current activity attract perception and must be processed to be taken into account.

When high resolution frames are fully filtered, segmented and analyzed, for example when using computer vision algorithms such as edge/feature point detectors or motion estimators, the high computational load required is obvious. Modeling retinal/cortical processes reproducing center surround ganglion cells or middle temporal (MT) cells behavior on wide receptive fields also introduces large-scale computations due to the variability of parameters such as speed, direction or receptor field size.

This article does not introduce a new technology nor algorithm, but combines different techniques and interfaces different programming languages to produce a portable generic library which can be used and extended by anyone without any knowledge in the optimization and graphics card programming fields. Though this library might be used as is, source code is provided not only to show how the wrapping of hardware specific functions is done, but also to convince the reader that good performance can be achieved at low cost and with little effort.

The library architecture consists of several layers manipulating filters. Filters not already implemented can be defined by the user in a standard language, then called from a script or a graphical user interface. The library then structures, orders and applies the filters on the fly depending on temporal and spatial dependencies. The lowest level layer can target central processing units (CPU) or graphics processing units (GPU) depending on available hardware. The library has been partly ported to support the Cell Broadband Engine and take advantage of its synergistic processing elements. The huge benefit from using vector instructions and such parallel ar-

chitectures is now affordable as the Cell BE is at the core of the Playstation®3 (installing Linux and the IBM SDK is however required). But since PCs with powerful graphics cards are widespread today, the shader implementation will be detailed.

Recent changes and extensions on graphics cards now allow us to implement parallel algorithms and use the GPU rather than the CPU for real-time applications and higher performances. Many companies and laboratories are investigating graphics hardware capabilities for general purpose computations but this article concentrates on how to easily make the most of it in a generic way for vision processes which share similarities with the original aim of graphics card pipelines. Though several libraries exist for general purpose computations using graphics hardware (GPGPU) such as RapidMind (previously known as Lib Sh)[9], Brook[6] or CUDA[2], they deal with data streams rather than pictures, integrate complex optimizations and include abstraction layers to be effective on any application. On the other hand, high level image processing frameworks such as CoreImage[1] or OpenVIDIA[12], neither generally provide access to intermediate results, handle time dependencies by default nor support arbitrary hardware. Of course many task specific GPU enabled applications have also been developed across the world [12,17,24], but this paper is an attempt to find a golden mean between generality and task dependence in the field of vision.

The portable implementation presented here is coded in Java 1.6 using JOGL **(Java bindings for OpenGL API)**, runs on low cost computers, allowing anyone to implement and apply spatial and temporal filters on any input source (computer generated or captured). Any intermediate output is stored as a texture and can be further processed or extracted on request. As long as the user does not ask for a transfer to the computer main memory, all computations are performed on the graphics card for speed improvement. Even if the hardware is in constant evolution (**render-to-texture** for instance is now widespread), the reader inclined to know the specificities and limitations of a GPU compared to a standard CPU might read the paper of Michael Shantzis on the subject[23]. An example of application in the field of retinal and cortical processes is detailed throughout this article. Although its purpose is neither to give an accurate description of biological processes nor to give a full account of how to model and optimize such processes, it illustrates the main capabilities of the library.

2 Theoretical framework integration

Fast visual processes can be useful in any framework **and a generic library needs to be compatible with all. The variability in vision theories is illustrated by the interactivist framework [5, 7], implying a radical shift in the requirements for a vision library.** Considering this approach, concepts and objects emerge from a network of interactions, each continuously an-

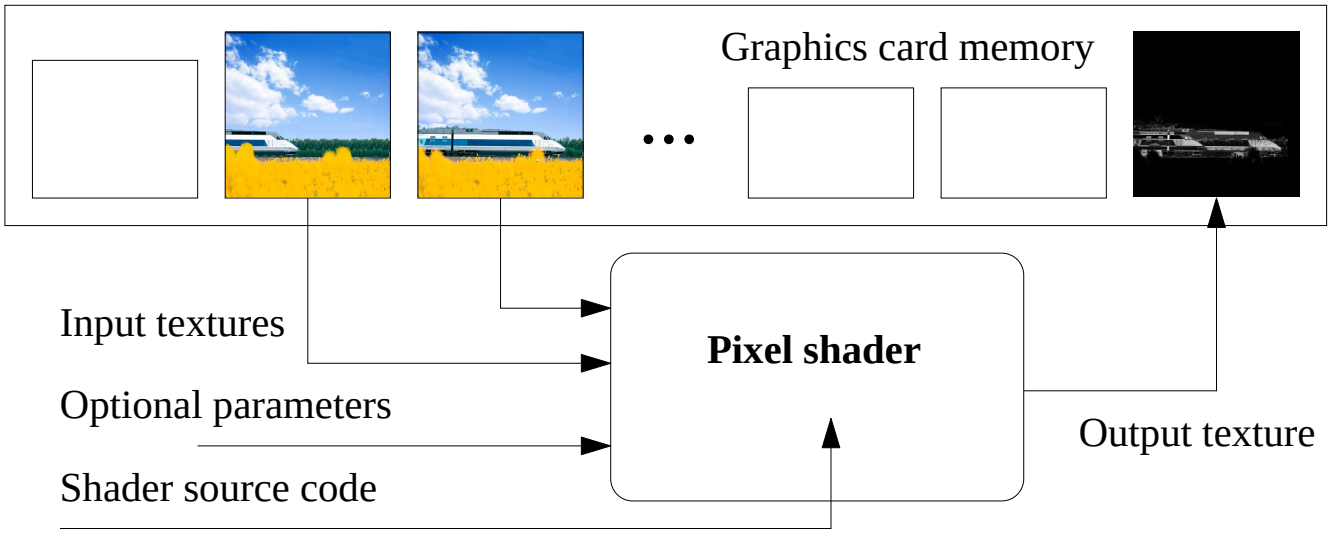


Fig. 2 Textures combination by a GPU pixel shader.

ticipating events resulting from taken actions. What is called events can be the activity of other internal processes or direct perceptions from various modalities. It is quite similar to Jean Piaget's sensori-motor theory [22] or Gibson's affordance theory [13,14]. More specifically when applied to vision, **it** shares many aspects of O'Regan's conclusions [20,21]. Local signals coming out of the optical nerve as well as higher-level features can be anticipated when eye saccades are performed. Though saccades might be considered as hindering the already uneven visual perception, several studies focus on the usefulness and intentionality of saccades [27,19]. For example when being exposed to a visual scenery, **a human** will recognize objects corresponding to correctly regulated interactions with mostly satisfied anticipations.

This kind of perception is normative : validity evaluation is relative within the set of learned concepts and does not require any homonculus to tell whether the matching is correct. If the context is unambiguous enough, there is no need for further exploration as soon as the activity in a highly connected part of the network, corresponding to a concept, starts to rise above the others. Moreover due to regulation and relative comparison, such processes are highly resistant to noise and easily adapt to variations in the environment. The resulting behavior also allows reducing the evoked potentialities space, concentrating on specific features which are contextually relevant [18]. For instance when walking in a crowd or driving in a dense traffic, human beings do not try to analyze the incredible amount of information that is available to them just by gazing around. A goal-oriented adapted behavior might consist in anticipating trajectories, avoiding obstacles, speeding up or slowing down, but certainly not in counting the number of persons dressed in white or reading every car plate, what might nevertheless be feasible [26].

Even if many recent theories [11,25] imply top-down modulations and avoid combinatory explosion when decomposing and evaluating the entire field of view, bottom-up processes are still needed to learn from and react to salient features in the environment. Additionally, low-level perceptive patterns are fast paced and less stable compared to abstract constructions of the mind[15]. Because of the high number of concurrent processes that might pick up information and match anticipations in the field of view, taking advantage of hardware optimizations to rapidly process parts of the sensors input should therefore be efficient.

In a neurobiological modeling perspective, though the visual cortex cells organization has been studied for a long time [16], reproducing vision processes on computer in real-time with a large input flow has a lot of advantages [3]. For instance, simulating MT-cells behavior may help to understand their origins and interactions with other layers of the cortex. More specifically, these are oriented speed detectors, only responding to a given range of directions and speeds. Their behavior can be interpreted as proto-anticipatory in that the spikes of each input cell are anticipated at different times, based on the MT-cell connections to lower cortex areas **(figure 1)**. It is quite robust to noise and allows the perception of speed even in complex motion [8]. Going on predicting blob movements relatively to actions rather than falling back to standard segmentation methods for higher abstraction levels might help to cope with problems encountered in real-time complex applications [4, 10].

Whatever may be a correct account of biological vision, easily testing hypotheses and adjusting parameters is of prime importance. By taking advantage of graphics cards computational power, one gets the opportunity to compare the results for different parameters values in real time. Moreover the library described in the next sections can also compile and reorganize filters on the fly, making it flexible enough to interactively and totally change filters source code and structure, avoiding the painful process of rebuilding an entire project. Finally, approaches based on perception dynamics or anticipation might benefit from a software framework where time dependencies are intrinsically managed.

3 Implementation

3.1 Single filter/process

All filters have the same generic structure allowing them to handle several input textures and produce a single output texture with arbitrary components. Consecutive computed output textures are stored in a cyclic buffer to avoid memory and time consuming copies, which size is computed based on filter dependencies. Figure 2 represents how textures are combined by a single GPU pixel shader program. The configuration of the graphics card and all OpenGL operations are hidden to the end user which only has to implement the shader program (us-

ing for instance GLSL = GL Shading Language) and to specify the input/output texture mappings. For example on the diagram, a differentiation is applied between the textures given as input. These input textures might be successive output frames from any filter, therefore also stored in a cyclic buffer. The associated shader source code could be :

```
uniform sampler2D tex0,tex1;
void main(void) {
    vec2 coords = gl_TexCoord[0].st;
    vec4 a = texture2D(tex0,coords);
    vec4 b = texture2D(tex1,coords);
    gl_FragColor = a−b;
}
```

Listing 1 Differenciation pixel shader source code. **All shader programs are written in GLSL. The so-called uniform variables are constant parameters for the shader that can be manipulated between executions, either by the card or the programmer.**

In the associated Java program controlling the execution of the filter, the shader is loaded on the graphics card and the textures are mapped to `tex0` and `tex1` using an `Effect` object. The texture mapping and source code can be easily modified during the execution of the Java program, giving a high flexibility for tests. `new Effect(n,w,h)` creates an effect that memorize `n` consecutive frames with dimensions (`w`,`h`). **The `difference` effect in listing 2** uses the `camera` effect output textures as input. This `camera` effect just stores raw input data in textures, allowing prefetching and optimal memory allocation. The last line maps the `camera` texture at current time `t` to `tex0` and the `camera` texture at discrete time `t-1` to `tex1`.

```
Effect camera = new Effect(2,w,h);
camera.setName("Input");
Effect difference = new Effect(1,w,h);
difference.setName("Differenciation");
difference.setShader("uniform [...] a−b; }");
difference.setTextures(
    new Effect[]{camera,camera},
    new int[]{0,1}
);
```

Listing 2 Differenciation at Java level.

Another example displaying additional capabilities of shader programs can be found in the annex. Though the understanding of given source code is not at all required to grasp the interest of combining high level languages (such as Java or C++) with shaders, it might be relevant for those who want to investigate deeper in the subject or generate their own shader programs on the fly at runtime.

3.2 Multiple processes

All features concerning effects sequences and temporal aspects will now be developed on a retinal/cortical pro-

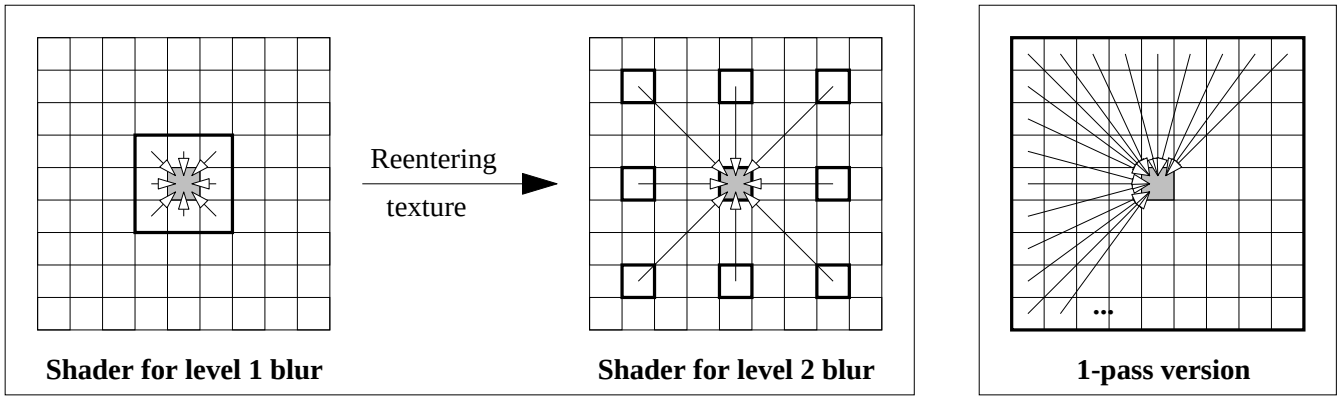


Fig. 3 Blur computation using either a single or 2-pass algorithm.

cesses set. As seen in previous section, every effect can be defined by its source code and the mapping of its input textures. Yet when applying several effects, their dependencies are to be taken into account. Though the source code is not included and some operations should be performed differently for optimization purpose, the set presented here has been chosen to show the possible interactions between effects.

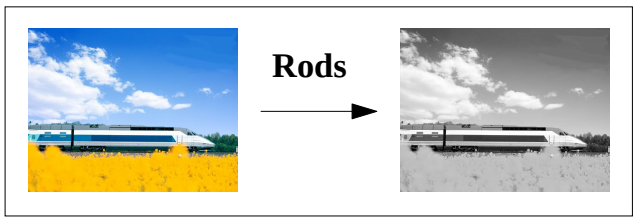


Fig. 4 Rods signal extraction from a "camera" input.

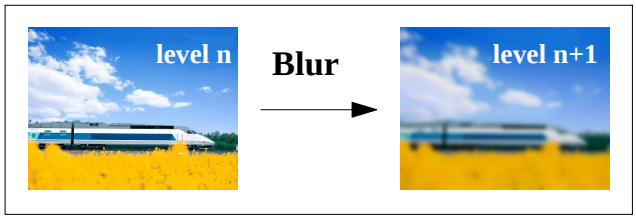


Fig. 5 Recursive additional blur.

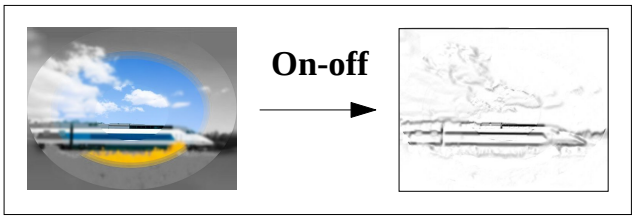


Fig. 6 On center off surround ganglion cells extracting dynamical contrasts.

The above figures as well as figures 8 and 9 represent effects. The name of the effect is written in the center and black arrows represent dependencies, linking standard input textures to the output texture produced by the effect. Most of the effects take only one texture as input as do the Rods (simulating retina rod cells, sensitive to a wide range of light wavelengths), On-off (reproducing on center off surround cells behavior based on contrasts) or Blur effects **(figures 4, 5, 6)**. The implementation of the Blur effect allows it to be recursively applied to change the blurring level either on all the retina for multi-resolution by increasing the size of the receptor fields, or on extra-foveal

areas to reproduce the increasing spacing between rods and cones.

Performing the blurring using a multiple pass algorithm allows not only to fasten the resulting texture generation by not repeating local computations, but also to provide a nice and easy way to obtain multi-resolution textures involved later in different scale object detection. A predictive dynamical system can for instance anticipate the movement or position of small features in the visual field as well as large objects occluding most of the scene.

Each pass is applied using a shader whose complexity is constant. As represented on figure 3, for every pixel in the texture, a mean value is computed on 9 pixels whether it is the first or second pass. By neglecting the texture borders, not causing any trouble on modern GPUs, we can approximate the complexity for a n-pass blurring on a $w \times h$ texture to $n \times w \times h \times 3^2$. This includes the generation of the n levels of blurring textures.

This can be compared to a straightforward approach computing the mean value on large receptor fields as shown on the right of figure 3 (9×9 pixel wide field equivalent to the result from the 2-pass algorithm). For the same parameters as in previous paragraph, the number of computation required would be $w \times h \times (3^n)^2$ (not even getting the intermediate resolutions).

To conclude with the different kinds of possibilities, textures from various effects can be merged into a single texture. **The Retina effect represented on figure 8** reproduces the loss of resolution and color on the periphery from the Rods and Blur effects outputs (without any distortion however). The last generated texture is also redirected as an input to simulate the retina persistence. Similarly, the MT effect computes local movements from a single input effect at different times **(figure 9)**. Though only 2 effects are involved, several textures are mapped to the inputs.

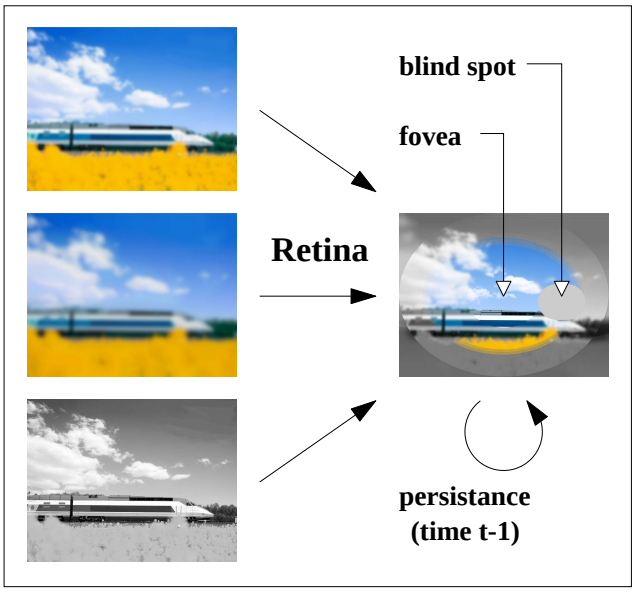


Fig. 8 Retina input simulation.

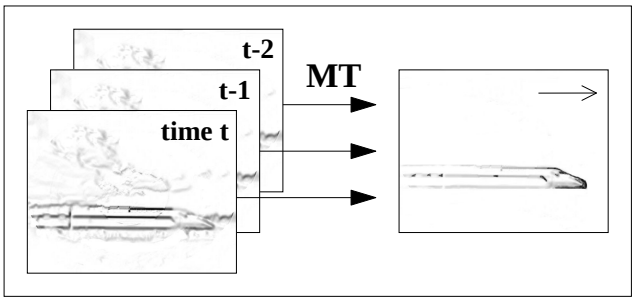


Fig. 9 MT cells sensitive to a **given** direction and speed.

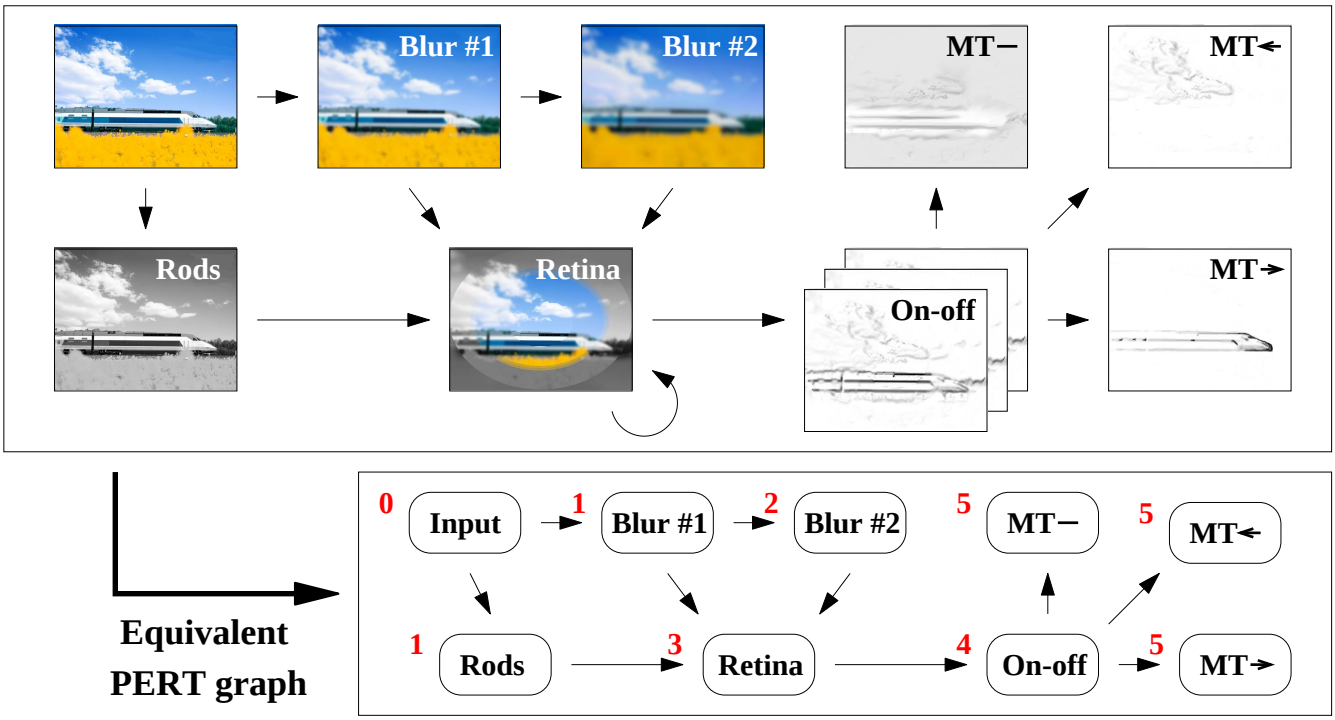


Fig. 7 Effects network to PERT graph conversion for shaders execution ordering. The red figures on the lower part correspond to the order of the computations. The lower the number, the earlier the process is run.

3.3 PERT ordering

The ordering of the shaders execution is obtained using a specialized PERT algorithm. The dependency graph is obtained using a single rule once all the effects have been configured independently. A dependency is introduced if an input texture is the output of another effect computed at current time (i.e. frame). Dependencies are represented as arrows on figure 7, the effect at the end of the arrow depending on the source. The cost for every dependency is set to 1, the goal being only to order the computations to avoid facing undefined input textures and to detect user graph conception mistakes.

When building the PERT graph, all dependencies with textures computed during previous execution cycles are excluded. For example on figure 7, MT only depends on On-off because of the use of its output at current time t, whereas there is no self-dependency for Retina, the persistence coming from the Retina output at time $t-1$.

Once the algorithm terminates without returning an error, the shaders are guaranteed to run correctly on well defined and updated textures. The number present next to an effect on the figure indicates how many effect layers are to be executed for it to perform successfully. Then for each update cycle, all shaders are therefore executed by following the ascending order on layers. The ordering within each layer is not relevant.

4 Performances

The tests have been performed on a laptop running Windows XP SP2 with an Intel Pentium M 735, 512Mo of DDR333 and a Mobility Radeon 9700. The 128Mo graphics card has been developed on a M11 architecture therefore possessing 4 pixel pipelines achieving 12 pixel shaders operation per clock cycle. Though this card is far less powerful than the graphics boards released in early 2008, the results using shaders are still impressive.

Using shaders, there are limitations on the number of operations in source code or textures used as input (texture unit mapping). Nevertheless they tend to vanish as the generations evolve or can be avoided by fractionating the shader program. The render-to-texture feature now available on most of the graphics cards has not been used to produce the following tables, hence limiting the theoretical performance.

4.1 Implementations comparison

On figure 11, performances are compared depending on the implementation. All programs, either implemented in C/C++ or Java, use the same architecture and filter graph composed of 10 effects computed on every frame. OpenGL computer generated 24 bit and 512x512 wide textures were set as inputs. The complexity or details present on the pictures do not impact on the performance since all pixels are processed the same way and the OpenGL frame generation is the same for the 3 implementations. Execution time is used for performance evaluation. The quite high standard deviations results from statistics computations and the chaotic calls to the garbage collector when the application is monitored. The C/C++ version has been developed to take into account possible differences with Java JOGL wrappers and evaluate the overhead and slowdown introduced by the Java language. Though there is a slight difference, it is not significative and cannot compete with the outstanding more than 100 times speed up of the shaders version. Such a speed up is a bit extreme but comes from the fact that all computations in the given example are easily made parallel in their integrality, even if they are representative of standard vision application needs. In case of hardware incompatibility, the CPU version might still be useful as a fallback for development purpose and later be swapped with a better implementation.

The comparison is not only provided to show the GPU version efficiency, but also to justify that the language choice for the filter source code has little impact on the overall performance, even when considering the possible transformations performed on it. Indeed, though a CPU overhead is introduced when generating, parsing or compiling the source file, the shader programs running on the GPU are not affected until the very last step, when loading the compiled program on the graphics card. This transfer might only happen once as long as the shader program source code remains unchanged, since it will not be uploaded again for simple parameter

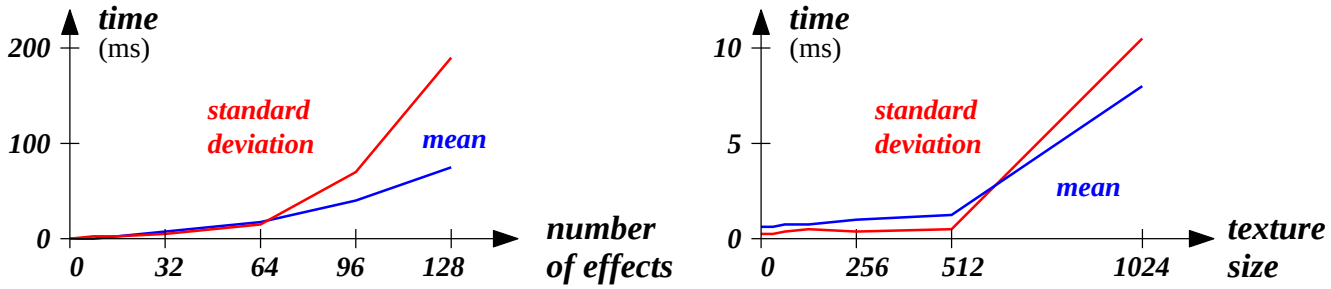


Fig. 10 Time performances depending on the number of effects to process for every frame and the size of the textures. **The mean values and standard deviations have been integrated over a large number of frame (≃1000).**

modifications. Therefore using graphics card native assembly language or any non portable low level language might only benefit to the expert programmer, but would not serve the purpose of any portable high level library.

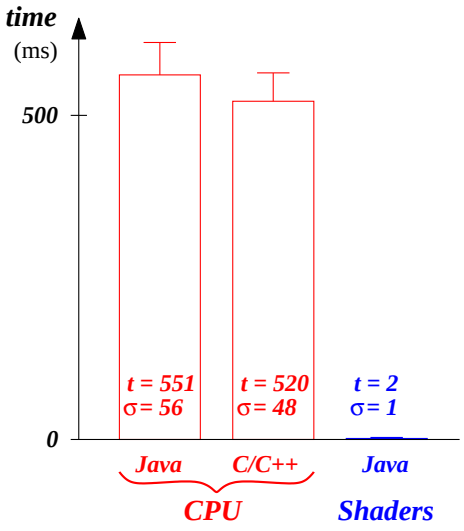


Fig. 11 Performance comparison between OpenGL implementations, using CPU or GPU Shaders for effects. Bars corresponds to mean times to update all the processes; error bars represent standard deviation.

4.2 Complexity

The first diagram on figure 10 displays results from computations on the same picture sequences and an increasing number of processes. The other one runs the same shader processes on larger and larger textures. The curves give the overall evolution and display the limitations relatively to the graphics card (memory and pixel pipelines). **In the case of extreme values for parameters, the high standard deviation results from recurrent and sudden huge variations around a relatively low mean value, due to memory transfers and reallocations. Still when** limiting the number of effects to the graphics card capabilities and inputting DVD video resolution, the evolution is almost linear.

Moreover when keeping parameters so as to avoid performance drop on some frames, the overall time cost of updating all the processes remains below 10 milliseconds. It allows the graphics card to perform additional computations to generate pictures or manipulate 3D models for example. Meanwhile the main processor is avail-

able for other tasks, turning this option into a good choice whenever an application requires massive parallel computations, even when they are not graphical.

5 Conclusion

Even if shaders are quite widespread today and the current paper focuses on this implementation of the library, similar development can be made on any multi-threaded or parallel architecture. Since such architectures are invading the personal computer market, this paper just tried to convince the reader that they can be transparent on specific tasks particularly well adapted to parallelism.

Hardware/software co-design or direct implementation on FPGA may be more and more flexible and provide cheap solutions for heavy computations, but they still required more knowledge than just standard programming skills. All the work done in the field of automatic parallelization and hardware abstraction is simply amazing but still not intuitive or mature enough to be widespread. Though the presented library is neither a general solution nor an optimal one, it has been used since November 2006 in our laboratory for low-level vision processes simulation. This narrow and task specific approach however allows the user to easily manipulate high level objects such as picture sequences, transform them and combine them in real time through a complex but intuitive relation based graph structure. The library therefore efficiently combines existing techniques, namely general purpose computations on GPU, code and context swapping on the GPU, cyclic buffering and PERT ordering.

Annex

In this section are gathered pieces of code which can greatly help in understanding or developing a similar library. For simplicity and explanatory reasons, the graph generation and process ordering algorithms are not detailed. For the same reasons, all interactive graphical user interfaces to add, modify or remove filters on the fly are not apparent. Additional functions are of course necessary to generate or capture the original input frames as well as to display the results, but these aspects are out of the scope of this paper.

The first Java source fragments provided on listings 3 and 4 define functions to produce shader source strings. This way, effects can be parameterized from the Java source code without any intervention on external files (external files may still be imported if provided by third party developers). Such parameters range from the direction and speed for the MT simulation to the range for the blurring. These samples do not take advantage of the shaders capability to handle additional uniform variables, making it possible to modify the process without reloading the program (see listing 1 for a definition). Nevertheless

the fast generation and upload of the program using Java allows to totally change the sequence of instructions and test more complex variations. The generated filters are simplified compared to more biologically realistic processes. They for example do not take into account the integration of neuron inputs over a time window.

```
public String sourceRetinaBlur(float dist, int radius) {
    String s =
    "uniform sampler2D tex0;\n" +
    "void main(void) {\n" +
    "    vec2 texCoord = gl_TexCoord[0].xy;\n" +
    "    const vec2 center = vec2(0.5,0.5);\n" +
    "    float dist = length(texCoord-center);\n" +
    "    const float offset = 1.0 / 512.0;\n" +
    "    vec4 color = texture2D(tex0, texCoord);\n" +
    "    if (dist>" + dist + ") {\n";
    for (int i=-1; i<=1; i++) {
        for (int j=-1; j<=1; j++) {
            if (i!=0 || j!=0)
                s +=
                "        color += texture2D(tex0, " +
                "texCoord + vec2(" + radius*i +
                ".* offset, " + radius*j + ".*offset ));\n";
        }
    }
    return s +
    "    color /= 9.;\n" +
    "    }\n" +
    "    gl_FragColor = color;\n" +
    "}";
}
```

Listing 3 Java code to return the shader code for "blurred pictures".

```
public String sourceMT(int dx, int dy) {
    return
    "uniform sampler2D tex0, tex1;\n" +
    "void main(void) {\n" +
    "    const float offset = 1.0 / 512.0;\n" +
    "    vec2 texCoords = gl_TexCoord[0].st;\n" +
    "    vec4 a = texture2D(tex0, texCoords);\n" +
    "    vec4 b = texture2D(tex1, texCoords + vec2(" +
        (float)dx + "*offset, " + (float)dy +
        "*offset ));\n" +
    "    if (length(a)>0.1 && length(b)>0.1)\n" +
    "    gl_FragColor = vec4(1.0-10.0*clamp(
        length(a-b),0.0,0.1));\n" +
    "    else\n" +
    "    gl_FragColor = vec4(0.0);\n" +
    "}";
}
```

Listing 4 Java code to return the shader program for "MT-cell pictures" as a string.

The previous functions can be called to generate shader source code on-the-fly but it needs to be compiled and executed by an `Effect` object

to be effective. The Java fragment of listing 5 defines several filters and their relations. By updating the effects in the right order (in this case, the ascending order on the array index), all filters can be applied for each incoming frame. **Using the library via the `Effect` object, the user only needs to define the various bindings between the textures and which programs need to be applied. Any change in this code will be reflected by different intermediate and output textures. Displaying arbitrary generated textures can be performed by rendering them on an arbitrary OpenGL primitive. Although they are not reproduced in this paper, the library also provides abstractions for this aspect, facilitated by the recent integration of JOGL with the standard Java2D graphics package.**

```
int [][] speed = new int[][]{
    { 2, 2},{-2, 0},{ 2, 0},{ 0,-2},
    { 0, 2},{-1, 0},{ 1, 1},{ 0, 1}
};
float [] blur = new float[]{0.1f, 0.2f, 0.3f};

Effect.setGL(gl);
effects = new Effect[2+blur.length+speed.length];

effects [0] = new Effect(1,width, height);
effects [0]. setName("Input");
int off = 1;
for (int i=0; i<blur.length ; i++) {
    int index = off+i;
    effects [index] = new Effect(1,width, height);
    effects [index].setName("Retina blur " + index);
    source = sourceRetinaBlur(blur[i], i+1);
    effects [index].setShader(source);
    effects [index].setTextures(new Effect[]{
        effects [index-1]},new int[]{0});
}
off += blur.length;
effects [off] = new Effect(2,width, height);
effects [off].setName("On-off cells");
source = sourceOnOff();
effects [off].setShader(source);
effects [off].setTextures(new Effect[]{
    effects [off-1]},new int[]{0});
off++;
for (int i=0; i<speed.length; i++) {
    int index = off+i;
    effects [index] = new Effect(1,width, height);
    effects [index].setName("MT cells");
    source = sourceMT(speed[i][0],speed[i ][1]);
    effects [index].setShader(source);
    effects [index].setTextures(new Effect[]{
        effects [off-1], effects [off-1]},new int[]{0,-1});
}
```

Listing 5 Main **java** source code configuring the effect pipeline.

Finally, though the above lines are all the user must be concerned with when configuring filters and combining them, **the listing 6 is provided** to show how the computations are made on the graphics card and the textures organized. **The detailed and commented functions are the central part of the library, demonstrating that a few lines of code are enough to parallelize complex operations on 2D discrete streams.**

The full library will be distributed as open source. A package as well as a detailed technical tutorial should be available on the author's research webpage (`http://quinton.perso.enseeiht.fr`**). Do not hesitate to contact the author in case of unavailability.**

6 Acknowledgments

We gratefully acknowledge the support of the National Institute of Informatics (Tokyo, Japan) as well as the third year students from the computer science and applied mathematics department from ENSEEIHT (Toulouse, France) for the Cell-BE porting.

```
//-------------------------------------------------------------------
//---------------------- Effect class (fragments) ---------------------
//------------------- shader program configuration and execution ------------------
//-------------------------------------------------------------------

Effect [] in_texs; // input source effects (dynrather than directly referencing the textures)
int [] in_times; // relative time to processed "frame" for each input texture (0,-1,...)

int [] out_texs; // array of generated output texture (length = time window required)
int out_current = 0; // current time index (frame in the cyclic buffer)
int width, height; // dimensions of the output texture

int shader = -1; // no associated shader at initialization (referenced by a program index on the card)

/** Constructor */
public Effect(int time_window, int width_, int height_) {
    width = width_; // width of the output texture
    height = height_; // height of the output texture
    // Generate the new textures (effect output)
    out_texs = new int[time_window]; // cyclic buffer of indexes to access the textures
    gl.glGenTextures(time_window,out_texs,0); // allocate texture indexes on the graphics card
    for (int o : out_texs) { // configure the texture OpenGL object (color, size, transformations...)
        gl.glBindTexture(GL.GL_TEXTURE_2D,o); // define the current texture index
        gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST);
        gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);
        gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, GL.GL_CLAMP);
        gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_T, GL.GL_CLAMP);
        gl.glTexImage2D(GL.GL_TEXTURE_2D,0,GL.GL_RGBA8,width,height,0,GL.GL_RGB,GL.GL_FLOAT,null);
```

```
    }
}

/** Set the associated shader program */
public void setShader(String source) { // the input string is the GLSL source code
    shader = gl.glCreateProgramObjectARB(); // allocate a new program index
    // Create the fragment program (through the pixel pipeline)
    int fshader = gl.glCreateShaderObjectARB(GL.GL_FRAGMENT_SHADER_ARB); // index for compilation
    gl.glShaderSourceARB(fshader,1,new String[]{source},null,0); // define the source code
    gl.glCompileShaderARB(fshader); // compile the source code
    int [] compiled = new int[1]; // array needed to store the compilation status with OpenGL bindings
    gl.glGetObjectParameterivARB(fshader, GL.GL_OBJECT_COMPILE_STATUS_ARB, compiled, 0); // status?
    if(compiled[0]==GL.GL_FALSE) { // could not compile
        System.out.println("Shader '"+ name + "' could not be compiled\n"); // notice the user
        int [] length = new int[1]; // array to get the log length
        gl.glGetObjectParameterivARB(fshader, GL.GL_OBJECT_INFO_LOG_LENGTH_ARB, length, 0);
        if (length[0]>1) { // additional information is available
            byte[] info_log = new byte[length[0]]; // buffer to store the text
            int [] info_length = new int[1]; // array to store the effective length read
            gl.glGetInfoLogARB(fshader, length[0], info_length, 0, info_log, 0); // read info
            System.out.println("GLSL Validation >> " + new String(info_log)); // display info to the user
        }
        shader = -1; // the shader was not correctly loaded and no functionel shader index is defined
    } else {
        gl.glAttachObjectARB(shader,fshader); // attach the shader object to the program
                gl.glLinkProgramARB(shader); // link the shader on the graphics card
                int [] progLinkSuccess = new int[1]; // array to store the linking status
                gl.glGetObjectParameterivARB(shader,GL.GL_OBJECT_LINK_STATUS_ARB,progLinkSuccess,0);
```

```
        if (progLinkSuccess[0]==GL.GL_FALSE) { // could not link
            System.out.println("Shader '"+ name + "' could not be linked\n"); // notice the user
            shader = -1; // the linking was not succesful, no functionel shader index is defined
        }
    }
}

/** Switch the texture to next time/frame within the cyclic buffer */
public void switchTexture() {
    out_current = (out_current+1)%out_texs.length; // only changing the index avoids memory transfers
}

/** Execute the shader program and stores the result in graphics memory */
public void execute() {
    // switch to next frame for this effect (another reason why the ordonancing is necessary)
    switchTexture();
    // try to execute the shader to produce a new frame
    if (shader!=-1 && in_texs!=null) { // are the inputs and the shader correctly defined?
        gl.glUseProgramObjectARB(shader); // activate the shader program
        boolean error = false; // boolean to store if binding errors happen with the input textures
        for (int i=0; i<in_texs.length; i++) { // run through all input textures
            int shader_in_tex = gl.glGetUniformLocationARB(shader, "tex" + i); // get the sampler uniform variable
            if (shader_in_tex==-1) { // an error occured, it is impossible to bind the texture
                System.out.println("Can not get the parameter : tex" + i + "\n"); // notice the user
                error = true; // we may stop without executing the program (inputs are incorrect)
            } else { // we can set the input texture
                gl.glActiveTexture(GL.GL_TEXTURE0+1+i); // activate the correct texture in graphics memory
                gl.glBindTexture(GL.GL_TEXTURE_2D, in_texs[i].getTexture(in_times[i])); // bind texture & input
```

```
            gl.glUniform1iARB(shader_in_tex,1+i); // bind input & variable
        }
    }
    if (!error) { // all inputs could be bound correctly
        gl.glBegin(GL.GL_QUADS); // draw a "screen quad" to go through the pipeline and apply the effect
        gl.glTexCoord2f(0, 0); gl.glVertex3f(−1, −1, −0.5f);
        gl.glTexCoord2f(1, 0); gl.glVertex3f( 1, −1, −0.5f);
        gl.glTexCoord2f(1, 1); gl.glVertex3f( 1,  1, −0.5f);
        gl.glTexCoord2f(0, 1); gl.glVertex3f(−1,  1, −0.5f);
        gl.glEnd();
    }
    // disable the shader program (so other operations using the pipeline can be performed normally)
    gl.glUseProgramObjectARB(0);
  }
  // finally copy the generated frame buffer into the "current time" output texture
  // (if the textures were not correctly bound, we initialize the output with a black picture)
  gl.glActiveTexture(GL.GL_TEXTURE0); // activate the output texture
  gl.glBindTexture(GL.GL_TEXTURE_2D, getTexture(0)); // bind the output with the texture
  gl.glCopyTexSubImage2D(GL.GL_TEXTURE_2D,0, 0, 0, 0, 0,width,height); // copy data
  // (the cost of the copy can be avoided with the render−to−texture option of recent cards)
}
```

Listing 6 Effect java class source code fragments handling the shader program configuration and execution **in java**.

References

1. CoreImage on Apple website. `http://developer.apple.com/macosx/coreimage.html`.
2. CUDA Zone on NVidia website. `http://www.nvidia.com/object/cuda_home.html`.
3. D. Bálya and B. Roska. Retina model with real time implementation. In *ISCAS*, volume 5, pages 5222–5225, 2005.
4. J.-L. Basille, J.-C. Buisson, and J.-C. Quinton. Interactivist navigation. In *European Conference on Cognitive Science (EuroCogSci'07)*. Lawrence Erlbaum Associates, 2007.
5. M. H. Bickhard. The emergence of representation in autonomous embodied agents. In *AAAI Fall Symposium on Embodied Cognition and Action*, 1996.
6. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.
7. J.-C. Buisson. A rhythm recognition computer program to advocate interactivist perception. *Cognitive Science*, 28(1):75–87, 2004.
8. C. W. Clifford, S. A. Beardsley, and L. M. Vaina. The perception and discrimination of speed in complex motion. *Vision Research*, 39, 1999.
9. M. D. M. R. Co.). A unified development platform for Cell, GPU, and multi-core CPUs. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC07)*, 2007.
10. J. Díaz, E. Ros, S. Mota, G. Botella, A. Cañas, and S. Sabatini. Optical flow for cars overtaking monitor : the rear mirror blind spot problem. *Ecovision (European research project)*, 2003.
11. A. K. Engel, P. Fries, and W. Singer. Dynamic predictions: Oscillations and synchrony in top-down processing. *Nature Reviews*, 2, 2001.
12. J. Fung and S. Mann. OpenVIDIA: parallel GPU computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM.
13. J. J. Gibson. *Perception of the Visual World*. Houghton Mifflins, Boston, 1950.
14. J. J. Gibson. *The ecological approach to visual perception*. Houghton Mifflins, Boston, 1979.
15. J. Hawkins and S. Blakeslee, editors. *On Intelligence*. Times Books, 2005.
16. D. Hubel and T. Wiesel. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology*, 160:106–54, 1962.
17. S. J. Kim, D. Gallup, J.-M. Frahm, A. Akbarzadeh, Q. Yang, R. Yang, D. Nistér, and M. Pollefeys. Gain adaptive real-time stereo streaming. In *International Conference on Computer Vision Systems*, 2007.
18. G. Kuhn and B. W. Tatler. Last but not least - magic and fixations : Now you don't see it, now you do. *2005*, 34, Perception.
19. D. MacKay. Visual stability and voluntary eye movements. *Handbook of sensory physiology*, VII/3A:307–331, 1973.
20. J. O'Regan. Solving the "real" mysteries of visual perception : the world as outside memory. *Canadian Journal*

of Psychology, 46:461–488, 1992.

21. J. O'Regan and A. No. A sensorimotor account of vision and visual consciousness. *Behavioral and Brain Sciences*, 24, 2001.
22. J. Piaget, editor. *The Origins of Intelligence in Children*. International Universities Press, 1952.
23. M. A. Shantzis. A model for efficient and flexible image computing. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 147–154, New York, NY, USA, 1994. ACM.
24. S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 2007.
25. L. M. Vaina, J. Solomoni, S. Chowdhury, P. Sinha, and J. W. Belliveau. Functional neuroanatomy of biological motion perception in humans. *PNAS*, 98, 2001.
26. J. P. Wann and R. M. Wilkie. How do we control high speed steering? *Optic flow and beyond book contents*, pages 401–419, 2004.
27. R. M. Wilkie and J. P. Wann. Eye-movements aid the control of locomotion. *Journal of Vision*, 3:677–684, 2003.