Numerical Methods for Obtaining Multimedia Graphical Effects

Alexandru SMEUREANU[1], Ștefan Daniel DUMITRESCU[2]
[1]Academy of Economic Studies, Bucharest,
[2]Politehnica University of Bucharest
alexandru.smeureanu@i-neo.ro, dumitrescu.stefan@gmail.com

This paper is an explanatory document about how several animations effects can be obtained using different numerical methods, as well as investigating the possibility of implementing them on very simple yet powerful massive parallel machines. The methods are clearly described, containing graphical examples of the effects, as well as workflow for the algorithms. All of the methods presented in this paper use only numerical matrix manipulations, which usually are fast, and do not require the use of any other graphical software application.
Keywords: *raster graphics, numerical matrix manipulation, animation effects*

1 Introduction

This article describes a few basic but very widespread visual effects than can be easily implemented. The methods described can be used as a first step towards understanding how different and more complex transformations can affect an image and why do they actually work. All the transformations and effects that mimic real world processes like fire and water reflections are created by respecting the common laws of physics.

Because each of the image instances of the animations are individually computed based on different mathematical functions, the animation can be regarded as a "vector animation" rather than normal animation that usually involves the fast redrawing of a sequence of raster images on the screen. Similar to the use of vector images instead of raster images, this technique shares some common properties: the size of the animation is considerably lower, it can be zoomed in or out without loss of quality and usually requires a larger amount of processor time to be displayed.

In the next part of this paper we will present the implementation of three effects: fire effect, water effect and the tunnel effect, better known as the wormhole effect.

2 Implementing the fire effect

By analyzing an infrared image of a real fire, we can observe how heat is distributed. Normally the heat is building up at the bottom of the fire and is gradually dissipating towards the top of the fire. Also, the shape of the fire flames tends to have a sharp triangle shape towards the top. Based on these observations we can conclude that the intensity of the fire in one point depends on the intensity of the fire bellow this current point. For implementation purposes, the intensity of the fire is measured with the use of an integer value between 1 and 255. Each intensity value is directly linked to a color value, obtained with the use of a gradient.

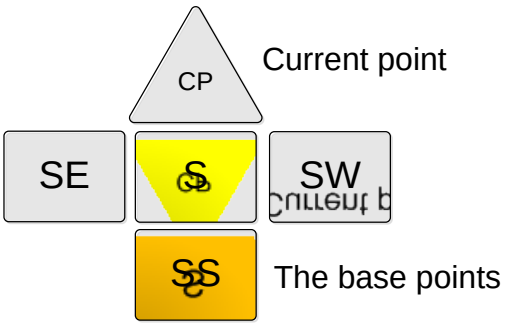

Fig. 1. Fire intensity modeling

$$I_{cp} = \frac{(I_{SE}*C_{SE}+I_S*C_S+I_{SW}*C_{SW}+I_{SS}*C_{SS})}{C_{CP}}$$

I_{SE} – Fire intensity on south east point
C_{SE} – Fire intensity on the south east point coefficient
I_{SW} – Fire intensity on south west point
C_{SW} – Fire intensity on the south west point coefficient

I_S – Fire intensity on south point
C_S – Fire intensity on the south point coefficient
I_{SS} – Fire intensity on south point further below
C_{SS} – Fire intensity coefficient on the south further point
C_{CP} – Attenuation coefficient
The intensity of all the points are computed and stored in a matrix with a number of rows and columns equal to the height and width of the desired image [1]. The intensity matrix is computed bottom up. The first row of the matrix has to be generated random or given by the user. This row represents the source of the fire.
The five intensity coefficients can be tweaked in order to create different looking fires [2]. In order to respect the symmetry of the fire, the south east and south west coefficients have to be the same. The most important coefficient of all is C_{CP} and represents the attenuation of the fire. If its value is too big the flames will be very short and if the value is too small, bellow the sum of all the other coefficients, the flames will leave the screen.
These are the results for two configurations:

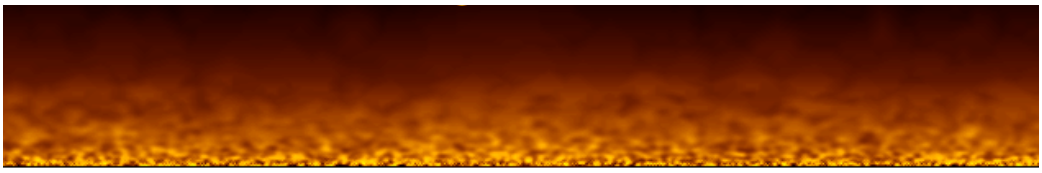

Fig. 2. Higher attenuation with random generated source of fire

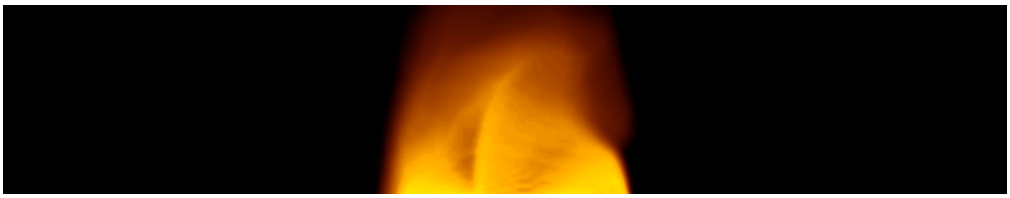

Fig. 3. Lower attenuation with still source of fire

Several other color improvements can be added to make it seem more real, like double gradient in order to simulate smoke.

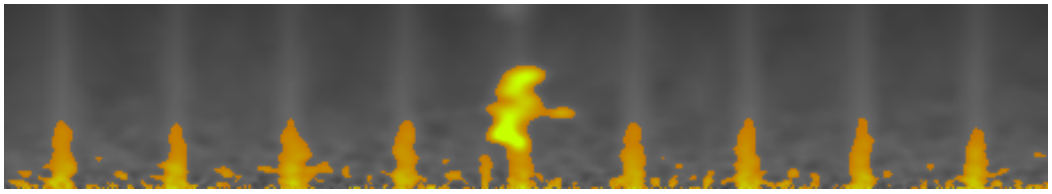

Fig. 4. Implementation of fire with smoke using double gradient

For an even more realistic effect, small random disturbances can be introduced in the matrix from time to time, and the fire source points can also be programmed to show a variance in time.

Fig. 5. Mechanical water waves

3 Implementing the water effect

Water waves are mechanical waves that travel on the surfaces of liquids. The implementation of a water effect is composed of three major parts [3]:

- The first part is the simulation of the wave propagation
- The second part is computing the light reflection
- Finally, rendering the image frame

Step 1. Simulating the water wave propagation

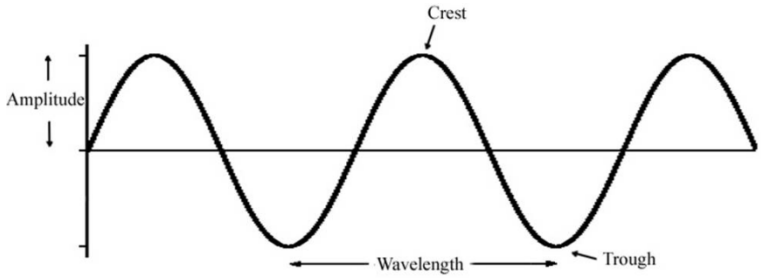

Fig. 6. Water wave components

Similar to the implementation of the fire effect, the water effect uses an intensity matrix which has the same dimensions as the image. The intensity matrix is initialized with zero values, representing calm water. In order to produce a wave, a point of the matrix is "pushed" by setting a negative intensity value.

The propagation is done by using a simple sampling algorithm very much similar to the algorithm used for blurring images. Current point intensities are computed based on the neighbors' values.

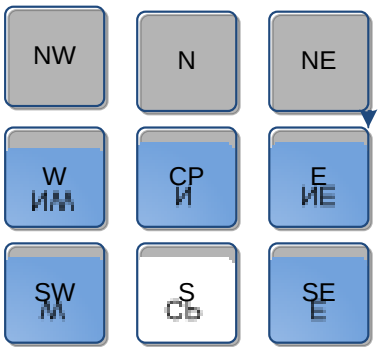

Fig. 7. Water wave intensity modeling

$$I_{cp} = \frac{(I_{SE} + I_S + I_{SW} + I_N + I_{NE} + I_W + I_E + I_{NW})}{AF}$$

where:

I_{SE} – Wave intensity on south east point
I_{SW} – Wave intensity on south west point
I_S – Wave intensity on south point
I_E – Wave intensity on east point
I_W – Wave intensity on west point
I_{NE} – Wave intensity on north east point
I_{NW} – Wave intensity on north west point
I_N – Wave intensity on north point
AF – Attenuation factor

Because the current point is computed based on all the neighbor points, the order in which the values of the points are computed affects the outcome. In order eliminate this effect an auxiliary matrix is used to store the results of the current stage. After all the values are computed, the auxiliary matrix is copied back over the intensity matrix.

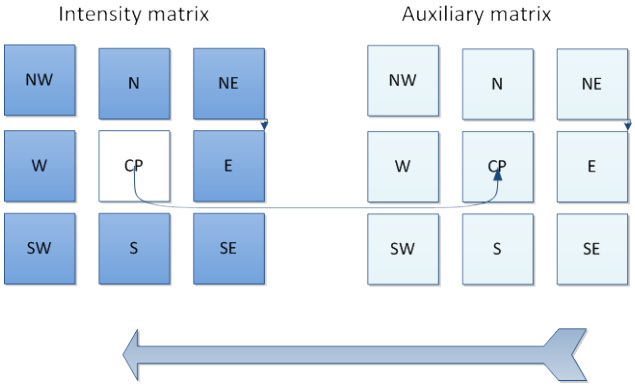

Fig. 8. The procedure for computing the intensity matrix

Step 2. Calculating the refraction of the light

Looking from above a water pool, object at the bottom will look skewed, due to water and air having different refraction indices. Refraction is the change in direction of a light beam due to a change in its speed. The change in speed is caused by the medium through which it passes.

This is a simple and well understood effect, and it is easy to calculate the exact light deviation.

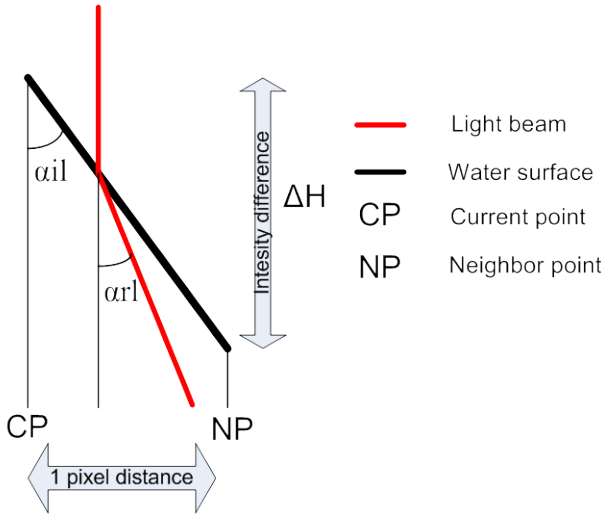

Fig. 9. Light refraction angle

The next step is to calculate the angle of incidence of the light. Because we are working in a 2D space, we have two different incidence angles for each axis:

$$tan \propto_{Xil} = 1/(CP - I_E)$$

In a similar manner the angle of light incidence for the Y axis is calculated:

In accordance to the laws of physics the light reflection index is:

$$I_R = \frac{\sin \propto_{il}}{\sin \propto_{rl}}$$

I_R – Light reflection index
α_{il} – Angle of incoming light
α_{rl} – Angle of reflected light

$$\propto_{rl} = \arcsin\left(\frac{\sin \propto_{il}}{I_R}\right)$$

$$tan \propto_{Yil} = 1/(CP - I_N)$$

Based on the angle of reflection and the height difference, we can calculate the displacement between what is directly at the bottom and what we actually see.

$$D = \tan(\propto_{rl}) * \Delta H$$
$$= \tan\left(\arcsin\left(\frac{\sin \propto_{il}}{I_R}\right)\right) * \Delta H$$

D – Displacement distance
ΔH – Height difference.

Step 3. Rendering the image

The next step is to calculate the displacements on the X axis and on the Y axis. The last step is to copy the background image used as bottom to the screen while applying the calculated displacements when copying each pixel.

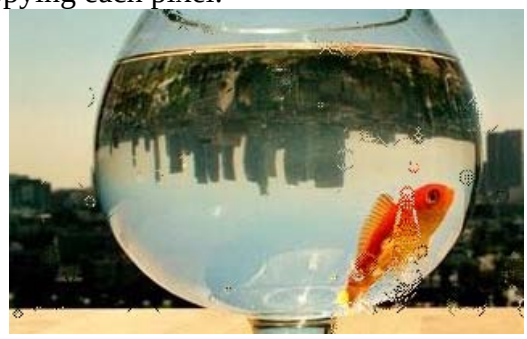

Fig. 9. The water wave effect result

4 Implementing the tunnel effect

The tunnel effect is a very interesting effect because although all graphic processing is done in two dimensional space the user is perceiving the animation as a three dimensional forward flight, in a tunnel that rotates.

The implementation of the effect is done in a way similar to the water effect: it needs a texture that will be applied on the walls of the tunnel by using several optical operations. In order to be efficient, the algorithm uses two additional matrices that will store precomputed displacements [4].

One of the matrices is an inverse distance table that contains the distance for every pixel of the screen to the center screen pixel. Because the pixels in the center of the screen are the furthest away, they will receive the highest values, while the ones close to the sides of the screen will receive lower values. This means that those points are closer to the camera. The second matrix contains the angles of each pixel on the screen relative to the center of the screen.

In order to be able to render the tunnel effect we need to use the two tables. This is done by copying the pixels from the texture to the output buffer and displacing them in accordance to the values of the pixel in the distance displacement matrix and the angle displacement value.

Fig. 10. Tunnel effect examples

Because the texture has a finite size and the depth of the tunnel tends to go towards infinite the texture has to be multiplied. This is done while the pixels are being applied to the tunnel by using the “modulo’ operator that will repeat the texture if the value gets out of bounds.

Fig. 11. Original textures used for the tunnel effect

The final versions of the formulas for computing the displacement are the following:
Formula for distance table initializations:

$$D_{x,y} = \left[\left\{ \frac{\left(R * T_H / \sqrt{(x - \frac{S_W}{2})^2 + (x - \frac{S_W}{2})^2} \right)}{T_H} \right\} * T_H \right],$$

where:
$[arg]$ – Integral out of argument
$\{arg\}$ – Fractional of argument
T_H – Texture image height
R – Tunnel depth factor. The tunnel depth is:
$R * T_H$
x – The column rank of the current pixel
$0 < x < S_w$
y – The row rank of the current pixel
$0 < y < S_H$
S_W – Stage width
Formula for angle table initialization:

$$A_{x,y} = \frac{T_W * atan2(y - \frac{S_H}{2}, x - \frac{S_W}{2})}{2 * \pi}, \text{ where:}$$

T_W – Texture image width
S_W – Stage width
S_H – Stage height
R – Tunnel depth factor. The tunnel depth is:
$R * T_H$

In the formula above we use the $atan2$ function because of several important features:
1. The one-argument arctangent function does not distinguish between diametrically opposite directions. For example, the anticlockwise angle from the *x*-axis to the vector <1, 1>, calculated in the usual way as $\arctan\left(\frac{1}{1}\right) = \frac{\pi}{4}$ (radians), or 45°. However, the angle between the *x*-axis and the vector <−1,−1> appears, by the same method, to be $\arctan\left(\frac{-1}{-1}\right) = \frac{\pi}{4}$ again, even though the answer clearly should be $-\frac{3\pi}{4}$, or −135°.
2. The normal arctangent method fails when required to produce an angle of $\pm\frac{\pi}{2}$ (or ±90°). For example, an attempt to find the angle between the *x*-axis and the vector <0,1> requires evaluation of arctan(1/0), which fails on division by zero. In contrast, $\arctan(1/0)$ gives the correct answer of $\frac{\pi}{2}$.
The $atan2$ function takes into account the signs of both vector components, and places the angle in the correct quadrant. Thus, $atan2(1,1) = \frac{\pi}{4}$ and $atan2(-1,-1) = -\frac{3\pi}{4}$.
In order to create the moving effect, we introduce two shifts variables when applying the texture to the stage: Δx and Δy. By increasing gradually the two variables we obtain two important animation effects. By increasing Δx we make the camera to move forward in the tunnel. By increasing Δy we make the camera rotate on its forward axis. Several other chaotic moving effects can be obtained by translating the central point of the tunnel.

Fig. 12. Chaotic tunnel axis shifting animation frames

After wrapping all of this together the formula for obtaining the color of each pixel based on the displacements:

$$Output_{x,y} = Texture\left[\left\{\frac{(D_{x,y} + \Delta x)}{T_w}\right\} * T_w\right], \left[\left\{\frac{(A_{x,y} + \Delta y)}{T_H}\right\} * T_H\right], \text{ where:}$$

- $[arg]$ – Integral out of argument
- $\{arg\}$ – Fractional of argument
- Δx – Animation shifting for moving ahead in the tunnel effect
- Δy – Animation shifting for tunnel rotation along the axis effect
- x – The column rank of the current pixel $0 < x < S_w$
- y – The row rank of the current pixel $0 < y < S_H$
- $Output_{x,y}$– Ouput image the x,y pixel
- $Texture_{x,y}$– The x,y pixel of the texture image initially provided

5 Possible hardware implementation

In the 1970s, NASA saw a need for computing power orders of magnitude beyond anything then available for satellite image analysis. Already operating at that time were satellites relaying voluminous information to Earth at high transmission rates, such as NASA's Earthscanning Landsat resources survey satellite sending digital data to ground stations at the rate of 15 million bits per second [5]. On the developmental horizon were satellites of far greater data gathering and transmission capacities. Because of the specific context, NASA commissioned the development of a new very small processor that came to be known as a Massive Parallel Processor. The aim of this project was to develop a small, simple and thus cheap processor that would be very easy to produce in large quantities. These processors would be interconnected in specific matrix like topologies in which each processor would be responsible for one pixel. In this way an image processing that usually took more than one day on a normal pixel by pixel processing approach, would now be solved in less than five minutes on a massive parallel machine.

A MPP machine based on the Blitzen processors has a SIMD (single instruction multiple data) architecture, meaning that at any given time all processors execute the same instruction but on a different set of data [6].

The abstract architecture of the MPP Blitzen processor is shown in figure 13.

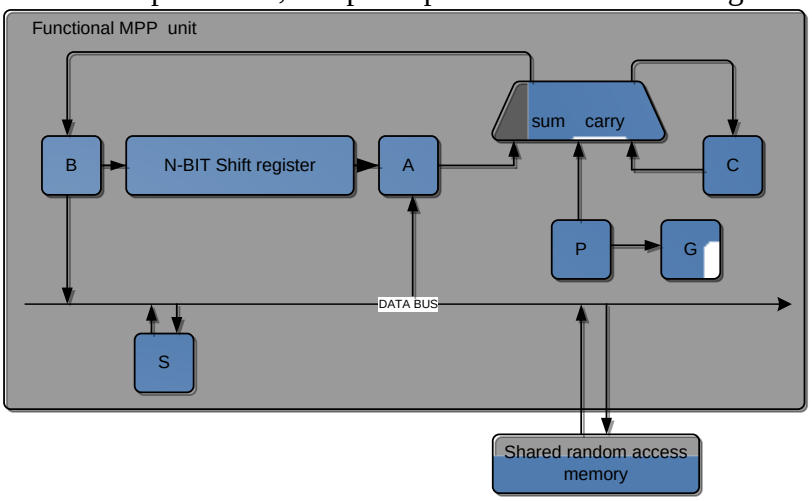

Fig. 13. Blitzen architecture

As it can be seen the internal architecture of a processor unit is very simple. In our proposed design, the processors are arranged into a two dimensional grid each processor having eight neighbors. In order to communicate with the others, the P register is used, along with a routing operation that will allow the value stored in register P to be transferred to the adjacent processor in its P register and the value from the opposite direction to be retrieved. This is a step-by-step, one instruction at a time, parallel processing.

This is the exact behavior needed to implement the fire effect. The other effects can also be easily implemented, having only

to add small procedures.
In the diagram from figure 14 we propose the implementation workflow of the fire effect.

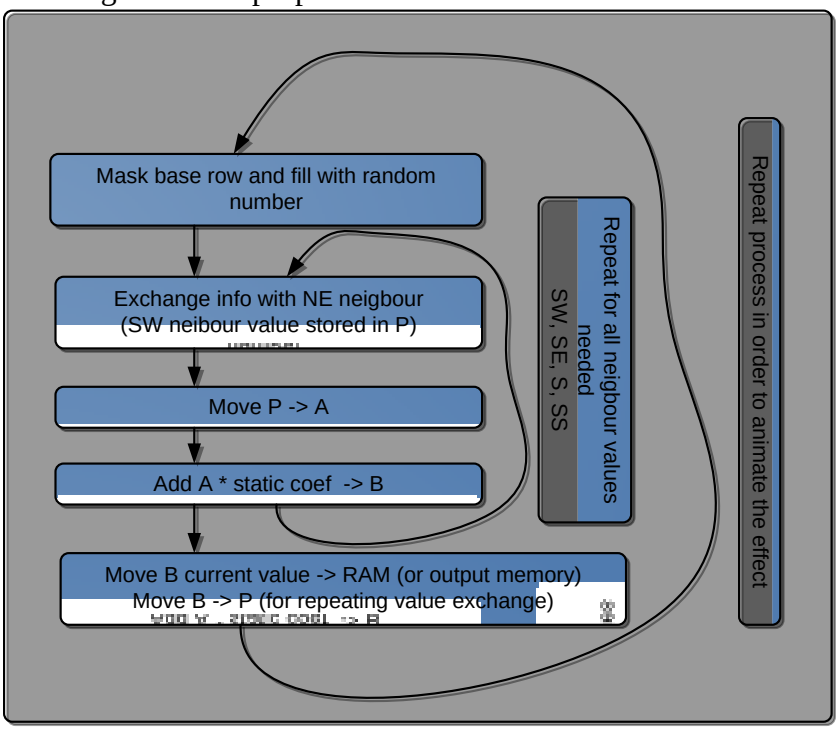

Fig. 14. Fire effect MPP implementation workflow

Implementing the water effect on the MPP machine differs a little from the implementation on a standard SISD (Single Instruction Single Data) machine such as the normal PCs. The main difference is that we don't need to have two data matrixes in order to ensure that already computed values in the current round will not affect the output of the rest of the pixels before ending the current round. This is because of the parallelism of the architecture that does all the pixel calculations in the same time, thus not allowing partial results to influence the outcome of the round, and also because each calculation is done locally in the processor's registers that are not shared.

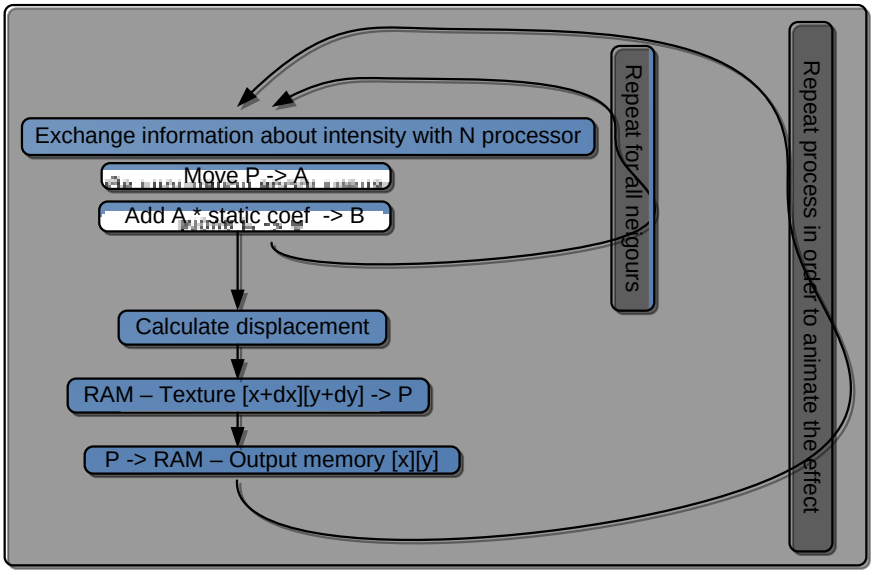

Fig. 15. Water effect MPP implementation workflow

Because the internal memory of the Blitzen processor is very small an external RAM memory is needed to store the texture and also the output.
Using simple and cheap processors can yield benefits when trying to implement an interesting graphical effect. Where the use of computers with complex CPUs is difficult, for example in creating lighting effects for shows, an embedded unit containing a few hundreds or even thousands of small simple processing units integrated in the image output device is a good choice.

6 Conclusions

There are lots of interesting animations effects present not only online or in computer games, but almost anywhere. Everywhere we look we can see eye-catching animations and effects used to lure people in and make them aware of a certain company product or to set a certain mood in a show by using light creatively. The cinema industry fully utilizes such effects, with great success [7].
Multimedia effects combined with words are known to have better impact in the learning process in contrast words alone [8]. Effects like the ones presented in this article can be used in physics lessons, like in elementary physics, to better visualize when explaining the mechanics of important topics such as: heat dissipation, wave propagation and optics.
Recent trends in application programming are going towards a new concept: (RIA) Rich Internet Application [9]. RIA applications are deployed online, downloaded and run locally in web browsers. They are designed to offer enhanced user experience even on slower PCs by using, among others techniques, vector animation [10]. Multimedia effects similar to the ones presented in this article are ideal for such applications because they don't consume much bandwidth, as they do not download images but actual machine code.
This article described the basic techniques that are needed to implement three simple effects: fire, water and the tunnel effect. Each of these effects needs only basic understanding of physics, and simple programming skills. Furthermore, these effects can always be improved upon by adding more complexity to the generation routines with relatively little effort [11].
The effects presented in this paper are almost entirely based on matrix operations, and as such are very well suited to be deployed on parallel architectures, here overlaid on the Blitzen system. An image projector connected to a small embedded device can provide a cheap way to create very interesting fire, water or tunnel effects [12].

References

[1] Academic Tutorials, Graphics Section, *Fire Effect*. Available: http://www.academictutorials.com/graphics/graphics-fire-effect.asp
[2] Processing.org, Tutorials, *Fire Effect*. Available: http://processing.org/learning/topics/firecube.html
[3] R. Willemse, *Game Development, Water Effect*. Available: http://www.gamedev.net/reference/articles/article915.asp, 2/15/2000.
[4] Academic Tutorials, Graphics Section, *Tunnel Effect*. Available: http://www.academictutorials.com/graphics/graphics-tunnel-effect.asp
[5] E.W. Davis and J. H. Reif, *Architecture and Operation of the BLITZEN Processing Element. 3rd International Conference on Computing on Supercomputing*, Boston, MA, May 1988.
[6] D.W. Blevins, E.W. Davis, R.A. Heaton and J. H. Reif, "BLITZEN: A Highly Integrated Massively Parallel Machine," *2nd Symposium on Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988, *Journal of Parallel and Distributed Computing*, Vol. 8, February 1990, pp. 150-160.
[7] F. T. Hofstetter and P. Fox, *Multimedia literacy*, New York, McGraw Hill, pp 77-90.

[8] R. E. Mayer, *The Cambridge handbook of multimedia learning*, Cambridge University Press, 2005, pp. 88-100.
[9] M. De David, *Web multimedia development.* Indianapolis SUA, New Riders Pub, 1996.
[10] Laszlo, "An Open Source Framework for Rich Internet Applications Laszlo," *An Open Source Framework for Rich Internet Applications*, 2005.
[11] T. Markas and John H. Reif, "Memory-Shared Parallel Architectures for Vector Quantization Algorithms," *1992 Picture Coding Symposium*, Lusanne Switzerland, March, 1993.
[12] M. Royals, T. Markas, N. Kanopoulos, J. H. Reif and J. Storer, "On the Design and Implementation of a Lossless Data Compression and Decompression Chip," *IEEE Journal of Solid-State Circ.*, Vol. 28, No. 9, pp. 948-953, Sep. 1996.

Alexandru SMEUREANU graduated Politehnica University of Bucharest, Automatic Control and Computers Faculty, Computer Science Department and Bucharest University of Economics, the Faculty of Cybernetics, Statistics and Economic Informatics. He is currently a PhD candidate in the field of Economic Informatics at University of Economics. His interests range in the software programming, network management, GIS systems, vector graphics and embedded devices programming.

Ştefan Daniel DUMITRESCU graduated Politehnica University of Bucharest, Automatic Control and Computers Faculty, Computer Science Department. He is currently a PhD candidate in the field of Semantic Technologies at Politehnica University of Bucharest. His interests range in the informatics applications with accent on online development, information extraction, knowledge representation and human-computer interaction. Among other skills, he is also interested in project management and skilled in network technology.