SoftwareX 9 (2019) 211–216

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Field Animation

N. Creati *, R. Vidmar, P. Sterzai

OGS-Istituto Nazionale di Oceanografia e di Geofisica Sperimentale, Borgo Grotta Gigante 42c, Sgonico, Trieste, Italy

ARTICLE INFO

Article history:
Received 18 September 2018
Received in revised form 16 January 2019
Accepted 25 February 2019

Keywords:
Python
OpenGL
Vector field
Animation

ABSTRACT

Vector fields visualization is quite common in many scientific disciplines and many methods have been developed for this purpose. Python is probably the ideal glue language to write applications that exploit the capabilities of modern graphic cards through the Open Graphics Library. A brief introduction to Python and OpenGL will preface our approach in writing easily reusable code to create a Python package to represent vector fields through particles that move along the flow lines of the field at a speed and color proportional to its modulus. An example application will be shown to illustrate how interactive control of speed, color and number of animated particles is possible as the whole rendering process happens in the GPU in real time. A background image can be shown to add information for the interpretation of the results.

Code metadata

Current code version	0.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2018_174
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Python, GLSL
Compilation requirements, operating environments & dependencies	Linux, Windows, Mac OS, Python, PyOpenGL, Numpy
If available Link to developer documentation/manual	https://bvidmar.bitbucket.io/fieldanimation/
Support email for questions	ncreati@inogs.it

0. Introduction

Visualization of vector fields is a well known problem when studying physical phenomena. Air flow around a wing, blood flow in a heart chamber, water circulation in an ocean, wind velocity, electromagnetic field are all examples of vector fields.

Vector fields can be visualized by texture advection [1] and geometry-based techniques. Among the former we can consider the spot noise [2] and line integral convolution (LIC) [3] methods. Examples of the latter are arrow/quiver plot [4], stream lines [5], particle tracing [6].

Texture-based methods create dense, continuous and uniform visualization of a vector field. They are cheap, fast and help users to recognize features in a flow. Geometry methods demand more computational resources and generally produce a coarse and sparse picture of the flow. The quality and details of geometry methods can converge to the texture-based representation if the number of tracing elements is dense enough [7].

The number of vector field grid points used in simulation increased by the time taking advantage of computer hardware capabilities but the maximum speed of a single CPU is by now at its physical limit (just above 3 GHz) due to cooling issues. The solution found to overcome this limit was the development of multi-core CPU. In the same time graphics processing have been delegated to GPU to distribute rendering among many dedicated cores. Traditional programming languages are not the best choice to profit by the level of concurrency offered by CPU and GPU architectures. MPI, OpenMP, CUDA and OpenCL were built to distribute work load among the cores but their usage is not straightforward and is generally restricted to IT specialists.

We describe here a lightweight and efficient package that we call 'Field Animation' (FA) for the visualization of 2D vector fields.

* Corresponding author.
E-mail address: ncreati@inogs.it (N. Creati).

https://doi.org/10.1016/j.softx.2019.02.008

FA shows a live animated picture of a vector field visualizing colored particles moving along flow lines, drawing pathlines that suggest the temporal evolution of the system in case of velocity vectors.

FA has been written in Python [8] and the visualization algorithm is based on OpenGL.

1. The python programming language

Python is an object oriented interpreted language and one of the preferred programming languages for data scientists for prototyping, visualization, and running data analyses on small and medium sized datasets. The main reasons of its success are that it is simple, free, friendly with other languages, object oriented and has a lot of available libraries. It is one of the most popular programming languages (4th according to the TIOBE Index, https://www.tiobe.com/tiobe-index/) and hundreds of tutorials can be easily found on the web. Python *Batteries Included Philosophy* means that the language is self-sufficient, comes out-of-the-box ready to use, with everything that is needed. Being interpreted it is inherently slow if compared with other compiled languages but its clear and simple syntax makes it ideal for scientist that cannot or will not rely on commercial, sometimes expensive applications. Programmers used to Fortran, C or C++ will be amazed by Python's default standard library (https://docs.python.org/3/library/index.html) richness. It contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Python is a dynamically typed language, completely object oriented. There is no need to declare variables before using them or declare their type. Every variable is simply an object bound to a name and unused (i.e. unbound) objects are automatically deleted by the garbage collection system of the interpreter. Every object is self documented and thanks to the introspective nature of Python, a complete list of a class or instance attributes and methods can be extracted at runtime. Any kind of documentation such as object descriptions or usage examples can be included in the code itself. Python organizes definition of variables and functions in script files called modules. All the objects defined in the modules can be imported in other script or modules to be reused. Python modules designed to work together are usually organized in packages with a hierarchical directory structure. At last the common theorem that interpreted languages are slow can be overcome by several packages that bring fast C or C++ compiled extensions to be accessible with the clear syntax of Python. Among them there are Numpy [9], a powerful package that offers fast arrays storage and calculus and Scipy [10] that provides signal, geometric, statistical and many other tools. Cython [11] allows creation of fast extensions with a Python like syntax as well as the wrapping of existing C libraries. The Python Package Index at https://pypi.org is a precious resource for free available packages that increase Python functionality and performance.

2. OpenGL

FA visualization algorithm is based on OpenGL [12]. OpenGL is a programming interface to the graphic hardware to produce 2D/3D scenes. It is a free and fast tool to create high quality images but it can be used to do some computation too. It is not a programming language but it provides an application programming interface (API) to the hardware to get the desired results as quick and efficient as possible. OpenGL provides a layer between the user application and the underlying hardware so that the program can work with no a priori knowledge of the graphic system and how it works or performs. The API controls the behavior of each pixel on the screen and hence the final visual results by managing all the steps needed to transform a 3D scene model (the graphic pipeline) defined by several OpenGL instructions to produce a two-dimensional image on the screen. The final output on the user display is the rendering of the defined 3D model. This library is used in CAD engineering and architecture applications, in computer-generated imagery (CGI), in the game and movie industry. It dates back to the beginning of the nineties and since then evolved to fulfill industry needs. Before 2004, the graphic pipeline was fixed and programmers could not modify the steps involved in the final rendering of a 3D scene. How screen pixels should behave was hard coded in the graphic hardware. After 2004 graphic cards switched from just rendering units to programmable ones with the advent of the OpenGL standard 2.0. The fixed graphic pipeline was abandoned in favor of the more dynamic programmable one. This change was favored by the introduction of a new class of graphic hardware, the Graphic Processing Unit (GPU). GPUs are highly parallelized and amazingly fast since they have many small programmable processors (the shaders cores) in charge to execute small programs (shaders). Shaders are written in a special high level compiled language, called GLSL (OpenGL Shading Language), that instruct some of the steps needed to render a scene. The number of programmable operations increased by the time and since the last version of OpenGL (4.5) shaders can also do computation operations out of the main rendering pipeline. At present days the programmable graphic pipeline is composed by some stages that are completely customizable by the user. The speed improvement depends on the execution of the rendering pipeline on a high parallel architecture; every programmable stage, defined by its shader, is executed on several shader cores (up to thousands in modern GPUs). Parallelization is not controlled by OpenGL: the hardware does the trick.

The building of a 3D scene is a complex operation since OpenGL provides only basic geometric tools. Every object in a scene must be defined by many vertices, represented by values of x, y and z in space, organized in geometric primitives such as points, lines or triangles and so the visualization of a complex scene can require the writing of hundreds of lines of code. OpenGL applications must be integrated in some windowing system in order to provide input, output and user interaction with the scene since the API takes only care of the rendering.

3. FA code design

Nowadays the visualization of large numerical models is tricky due to the high demand of graphical resources needed. Furthermore scientist cannot take care of the efficiency of both the numerical model and the visualization system. Commercial applications or free ones, like Visit [13] or Paraview [14], are often used to visualize data and understand their meaning. Unfortunately these applications provide only standard visualizations tools, like quivers or streamplots, and are slow or need datasets converted to specific formats. Field Animation was written during the development of the simulation package PyGmod [15] for the fast and accurate visualization of a geodynamic deformation field. The simulations are characterized by strong lateral viscosity and velocity changes due to the different rheological behavior of solid (rocks) and fluid (water or melt) materials. Throughout this paper we refer to version 0.1.

FA implements a particles tracing visualization algorithm [16] where particles move according to the field streamlines giving an instantaneous picture of its pattern and flow lines. This is a well known methods that suffers of the same issues of texture-based approaches when the number of seeding is too large and

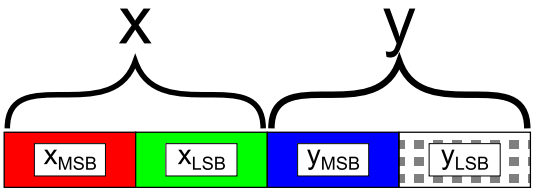

Fig. 1. Particle coordinates x and y are stored each in two of the four bytes (RGBA) of the texture point color.

the overlap of the points masks important information. Therefore FA has been developed so that the user can interact and change some parameters in real-time [17] to improve the visualization.

This Python package uses both PyOpenGL [18] and Numpy packages to take the most of the available hardware. The input vector field is a bidimensional array handled by Numpy and the rendering happens through OpenGL. Pure PyOpengl was preferred over other well known Python packages like VisPy [19], Mayavi2[20], ModerGL [21] or pyglet [22] because these high level interfaces introduce unavoidable limitations. PyOpenGL is a raw and transparent binding to the standard OpenGL C libraries. This approach lets the user free to embed FA in any other openGL based application as a widget. The adoption of the OpenGL programming pipeline [12] allows the usage of a large number of particles (millions) and hence more detailed models. Most of the computation is done by the GPU and it is instrumented by shaders written in GLSL. The CPU merely takes care of the initialization of some data structures and arrays. With this approach the programmer can focus on the dataset that will be imported using efficient Python libraries. FA has two support modules, *texture* and *shader*, to handle OpenGL textures and shaders code loading, compiling and linking. The *FieldAnimation* class takes care of arranging all the data, sets the right parameters to the OpenGL context for rendering the scene and defines the drawing workflow. The animated image is created instantiating the *FieldAnimation* class:

Listing 1: Animated image creation

```
animated_image = FieldAnimation(width, height, field)
```

The arguments *width* and *height* are the vertical and horizontal pixels dimensions of the window, *field* is an NxMx2 array with the field components.

All FA classes and functions have been kept as simple as possible to reduce dependencies but new functionalities can be easily added subclassing FA. The animated image created in listing 1 can be integrated in any windowing system.

3.1. Particle tracing algorithm

FA implements a simple OpenGL sequence of stages to draw on the screen: a vertex shader, a compute shader and a fragment shader. OpenGL connects these shader programs with fixed functions glue. In the drawing process the GPU executes the shaders piping their input and output along the pipeline until pixel will come out at the end. The vertex shader stage handles vertex processing such as space transformation, lighting and arranges work for next rendering stages. The fragment shader manages the stage after the rasterization of geometric primitive and defines the color of the pixel on the screen.

Particle tracing starts with the generation of an array of random particle positions on the screen. This array is stored in an OpenGL Texture object encoding them as colors (RGBA values). A 100 pixels $\times$ 100 pixels texture for example can store in this way 10.000 points positions. Particle coordinates are encoded into two bytes, RG for x and BA for y (see Fig. 1). Each texture pixel can therefore store 65536 distinct values for each coordinate. The texture is passed to the GPU in a vertex shader and the original particles positions are retrieved from the RGBA texture using the "texture fetched method" [12] in the vertex shader (Listing 2). Decoding of particles position from texture is implemented through an array with absolute indexes of the particles, passed to the shaders.

Listing 2: GLSL vertex shader code for the decoding of tracers positions from a RGBA texture.

```
    #version 430
    layout (location = 0) in float index;

    uniform sampler2D tracers;
    uniform float tracersRes;

    // Model-View-Projection matrix
    uniform mat4 MVP;
    uniform float pointSize;

    out vec2 tracerPos;

    void main() {
// Extracts RGBA value
vec4 color = texture(tracers, vec2(
        fract(index / tracersRes),
        floor(index / tracersRes)
        / tracersRes));

// Decodes current tracer position from the
// pixel's RGBA value (range from 0 to 1.0)
tracerPos = vec2(
        color.r / 255.0 + color.b,
        color.g / 255.0 + color.a);

gl_PointSize = pointSize;
gl_Position = MVP * vec4(
tracerPos.x, tracerPos.y, 0, 1);
    }
```

The core algorithm executes in the draw method of the FieldAnimation instance where the following operations occur at each step of the main rendering loop:

1. draw the modulus of the vector field or a user defined image if requested;
2. set a framebuffer texture (*screen texture*) as the main rendering target:
   (a) draw the *background texture* on the *screen texture* with a fixed opacity;
   (b) decode the particles positions from the *currentTracersPosition texture* and draw them on the *screen texture*;
3. set the rendering target to the active window;
4. draw *screen texture* on the active window;
5. swap *screen texture* and *background texture*;
6. calculate the new particles positions (in the update shader) and encode them in the *nextTracersPosition texture*;
7. swap *nextTracersPosition texture* and *currentTracersPosition texture*;

Step 6 happens entirely in the GPU taking advantage of its massive parallelism. GPU built before 2010 can do it only in a fragment shader while newer ones can take advantage of a dedicated compute shader. But there are more advantages in doing the update in a compute shader (Listing 3):

1. code is more readable, there is a clear separation between rendering and computational processes;
2. particles position decoding is done using the GLSL *imageLoad* function. This function retrieves for each pixel the exact value stored in the image while the *texture* function, commonly used in vertex or fragment shaders, gets an interpolated value from the nearest pixels;
3. the number of GPU cores used by the compute shaders can be tuned by the OpenGL *glDispatchCompute* function;
4. there is no need to allocate an extra framebuffer texture to store the new particle positions at each rendering step;
5. compute shaders increase performance up to 10%–20%.

Listing 3: Sample GLSL code from the implemented compute shader.

```
      void main() {
// Global index of work item processed by
// the compute shader
ivec2 uv = ivec2(gl_GlobalInvocationID.xy);

// Retrieves RGBA color from input image
vec4 color = imageLoad(tracers, uv);

// Converts color to current tracers positions
vec2 pos = vec2(
    color.r / 255.0 + color.b,
    color.g / 255.0 + color.a);

// Bilinear interpolate of field value for
// the tracer
vec2 velocity = mix(fieldMin, fieldMax,
    lookupField(pos));

// Normalizes field
float speed_t = length(velocity)
    / length(fieldMax);

// Calculates tracers offeset
vec2 offset = vec2(velocity.x, velocity.y)
    * fieldScaling * speed_factor;

// Updates tracer position, wrapping around
// the boundaries. Periodic boundary
// along x and y
if (periodic){
  pos = fract(1.0 + pos + offset);
}
else{
  pos = pos + offset;
}

// Define seed for radom tracers
vec2 seed = (pos + uv)  * rand_seed;

// Decay is a chance a tracer will
// restart at random position
// to avoid degeneration.
// Solve the problem of areas with fast
// points that are denser than areas
// with slow points.
// Increases reset rate for fast tracers
float new_seed = random_vec2(seed);
float decay = decay + speed_t
    * decay_boost;
float drop = step(1.0 - decay, new_seed);

// Creates new random tracer position
vec2 random_pos = random_vector(new_seed);
pos = mix(pos, random_pos, drop);
vec4 new_pos = vec4(fract(pos * 255.0),
    floor(pos * 255.0)/255.0);

// Saves the new tracer position in the
// output image
imageStore(resultImage, uv, new_pos);
    }
```

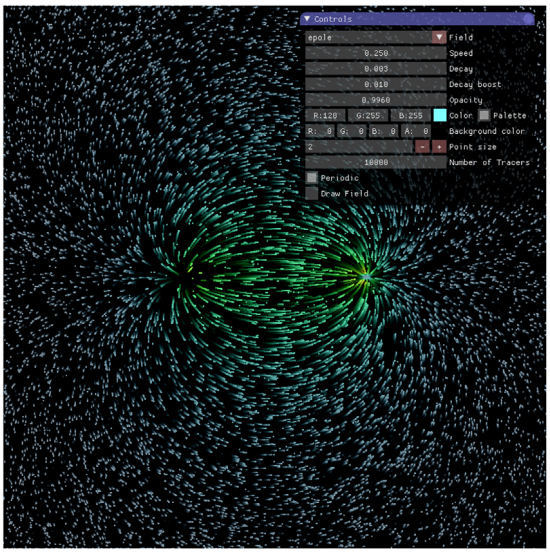

Fig. 2. Screenshot of the application provided in the *examples* directory of the package showing the electric field of a dipole.

During the calculation of new particles positions every particle P is moved to a new position $P_{new}(x_{new}, y_{new})$ computed using the vector field (interpolated from the given grid) and a new random point $P_{rand}(x_{rand}, y_{rand})$ is created. Then P will be replaced by P_{rand} if P_{rand} is "close enough" to P_{new}, otherwise by P_{new}. The distance used for the comparison is driven by a user selectable *decay* value that controls the progressive death of the particles. Since the GLSL language does not provide a native random number generator several algorithm have been tested. The adopted random generator model uses a simple integer hash function and inserts the result into a float mantissa [23]. This function creates a uniform distributed number in the range [0, 1]. Of course, the same field visualization could also be achieved using some alternative methods:

- store random points in a vertex buffer object (*VBO*) and update it at each rendering step by the Transform Feedback method [12];
- use the CPU for the computational part of the algorithm and the GPU only for rendering;
- write the whole algorithm using CUDA or OpenCL.

However all the above methods increase program complexity and probably decrease efficiency. The proposed algorithm could be further improved by the use of Uniform Buffer Object (*UBO*) introduced since OpenGL 3.1. All variables can be grouped in a structure that is assigned to the shaders attributes. An *UBO* makes the code cleaner and improves performance.

4. A working example

The creation of a FA image is straightforward: just instantiate the FA class passing the vector field array and call its draw

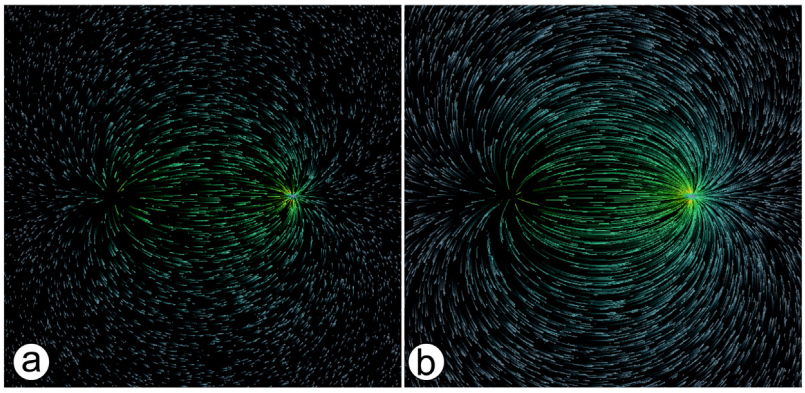

Fig. 3. Electric field of a dipole: effect of the speed parameter on visualization. (a) default speed 0.25; (b) speed 1.8. The particles move faster and the pathlines stretch.

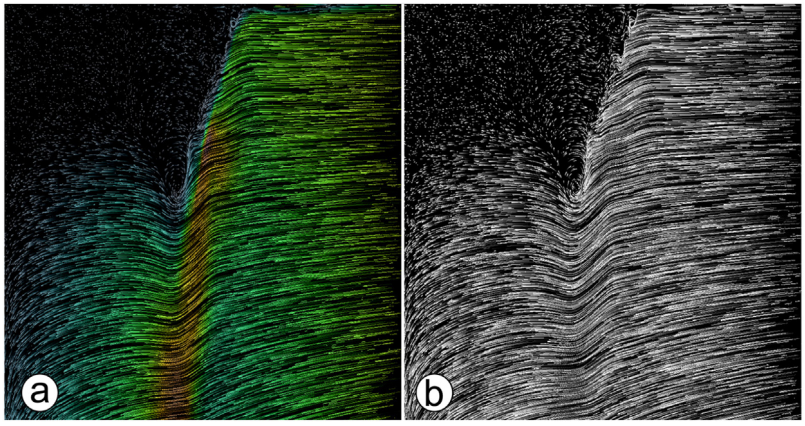

Fig. 4. Velocity field of a geodynamic simulation. (a) Color is field modulus in a cubehelix [24] colormap with parameters gamma = 0.9, minSat = 0.2, maxSat = 5.0, minLight = 0.5, maxLight = 0.9, startHue = 240.0, endHue = −300.0. The implemented cubehelix algorithm can use HSL color space value or just the standard start and rot switches. (b) Fixed color.

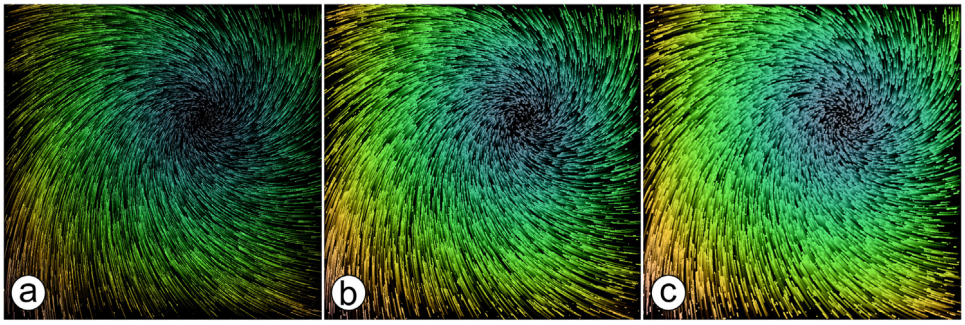

Fig. 5. Clockwise spiral field: different particles size. (a) Point size = 1; (b) Points size = 2; (c) Point size = 3.

method within the main rendering loop. An example (Fig. 2) to show how to use FA in a visualization application can be found in the *examples* directory. This application depends on two OpenGL packages: pyimgui [25] for setting interactively the visualization parameters and GLFW [26] for rendering the OpenGL image created by FA in a windowing system.

The interactive GUI in Fig. 2 allows to modify the visualization parameters that FA embeds as instance attributes (Listing 4):

Listing 4: FieldAnimation class attributes

```
# Default values
FieldAnimation.speedFactor = 0.25
FieldAnimation.decay = 0.003
FieldAnimation.decayBoost = 0.01
FieldAnimation.fadeOpacity = 0.996
FieldAnimation.color = (0.5, 1.0, 1.0)
FieldAnimation.palette = True
FieldAnimation.pointSize = 1.0
FieldAnimation.tracersCount = 10000
FieldAnimation.periodic = True
FieldAnimation.drawField = False
```

A detailed description of the parameters:

Speed: set the length of the field pathlines i.e. the speed of the particles: the higher the value the longer the particle trace length (Fig. 3);

Decay: set the life span of a particle;

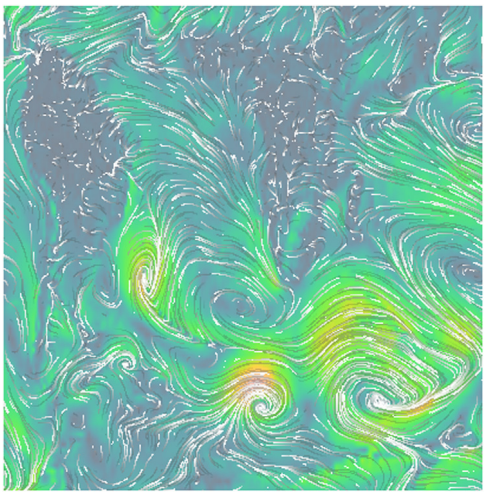

Fig. 6. Wind velocity modulus as background image with white particles traces.

Decay Boost: increase points density in low intensity field areas;

Opacity: set opacity of the particles over the background image;

Color: set color of the particles according to the field strength through a cubehelix based color map [24] (Fig. 4A);

Palette: set color of the particles to a constant value (Fig. 4B);

Point size: set the size of the animated particles (in pixels) (Fig. 5). The size depends also on the OpenGL *GL_VERTEX_PROGRAM_POINT_SIZE* flag. Values greater than 3 create weired effects. Custom markers can be used instead of points but this affects overall rendering speed;

Number of Tracers: set the number of the moving particles;

Periodic: if checked points that move outside the border of the rendering window will enter from the opposite one;

Draw Field: Draw the field modulus as a background image. In the example application the field modulus is rendered through a cubehelix color map (Fig. 6).

The **Field** combobox in Fig. 2 allows to choose among some synthetic vector fields implemented in the application and illustrated in figures from 3 to 6.

FA scales automatically the provided vector field to give the best visual effects. Speed and decay parameters help to fine tune the visualization and should be used for better understanding the physical phenomena.

5. Conclusions

FA implements a basic Python class to depict a vector field with an animated particles technique. It has been developed to show the deformation field occurring in geodynamic simulations but it has been generalized to visualize any bidimensional vector field. OpenGL takes cares of the rendering and the computational processes thanks to the programming pipeline. The overall performance of FA is not affected by the use of Python as the main programming language because most of the computation is done by the GPU and the remaining operations take advantage of Numpy methods.

Conflict of interest

The authors declare that they have no conflict of interest.

References

[1] Laramee Robert S, Hauser Helwig, Doleisch Helmut, Vrolijk Benjamin, Post Frits H, Weiskopf Daniel. The state of the art in flow visualization: Dense and texture-based techniques. Comput Graph Forum 2004;23(2):203–21. http://dx.doi.org/10.1111/j.1467-8659.2004.00753.x,

[2] van Wijk Jarke J. Spot noise texture synthesis for data visualization. In: Proceedings of the 18th annual conference on computer graphics and interactive techniques. ACM Press; 1991, http://dx.doi.org/10.1145/122718.122751.

[3] Cabral Brian, Leedom Leith Casey. Imaging vector fields using line integral convolution. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '93, New York, NY, USA: ACM; 1993, p. 263–70. http://dx.doi.org/10.1145/166117.166151, URL http://doi.acm.org/10.1145/166117.166151,

[4] Klassen RV, Harrington SJ. Shadowed hedgehogs: a technique for visualizing 2D slices of 3D vector fields. In: Proceeding visualization 91. IEEE Comput. Soc. Press. http://dx.doi.org/10.1109/visual.1991.175792.

[5] Kenwright DN, Mallinson GD. A 3-D streamline tracking algorithm using dual stream functions. In: Proceedings visualization 92. IEEE Comput. Soc. Press. http://dx.doi.org/10.1109/visual.1992.235225.

[6] Lane David A. UFAT: A Particle tracer for time-dependent flow fields. In: Proceedings of the conference on visualization '94. Los Alamitos, CA, USA: IEEE Computer Society Press; 1994, p. 257–64, URL http://dl.acm.org/citation.cfm?id=951087.951135,

[7] Weiskopf Daniel, Erlebacher Gordon. Overview of flow visualization. The Visualization Handbook 2005;261–78.

[8] van Rossum G. Python tutorial. Technical Report CS-R9526, Amsterdam: Centrum voor Wiskunde en Informatica (CWI); 1995.

[9] Walt Stéfan van der, Colbert SChris, Varoquaux Gael. The numpy array: a structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22–30.

[10] Jones Eric, Oliphant Travis, Peterson Pearu, et al. SCiPy. 2001. URL http://www.scipy.org/.

[11] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The best of both worlds. Comput Sci Eng 2011;13(2):31–9. http://dx.doi.org/10.1109/MCSE.2010.118,

[12] Woo Mason, Neider Jackie, Davis Tom, Shreiner. OpenGL programming guide. Addison-wesley Reading; 1999.

[13] Childs Hank, Brugger Eric, Whitlock Brad, Meredith Jeremy, Ahern Sean, Pugmire David, Biagas Kathleen, Miller Mark, Harrison Cyrus, Weber Gunther H, Krishnan Hari, Fogal Thomas, Sanderson Allen, Garth Christoph, Bethel E Wes, Camp David, Rübel Oliver, Durant Marc, Favre Jean M, Navrátil Paul. VisIt: An End-user tool for visualizing and analyzing very large data. In: High performance visualization–enabling extreme-scale scientific insight. 2012, p. 357–72.

[14] Ayachit Utkarsh. The paraview guide: a parallel visualization application. Kitware, Inc.; 2015.

[15] Creati Nicola, Vidmar Roberto, Sterzai Paolo. Geodynamic simulations in HPC with Python. In Kathryn Huff and James Bergstra (Eds.) Proceedings of the 14th python in science conference. 2015; p. 158–63.

[16] Fowler David, Ware Colin. Strokes for representing univariate vector field maps. CHCCS/SCDHM; 1989.

[17] Burger K, Kondratieva P, Kruger J, Westermann R. Importance-driven particle techniques for flow visualization. In: 2008 IEEE Pacific visualization symposium. 2008, p. 71–8. http://dx.doi.org/10.1109/PACIFICVIS.2008.4475461.

[18] Fletcher Mike C. PyOpenGL. Online available from: http://pyopengl.sourceforge.net/.

[19] Campagnola Luke, Klein Almar, Rossant Cyrille, Rougier Nicolas. VisPy. 2013. URL http://www.vispy.org/.

[20] Ramachandran Prabhu. Mayavi. 2008. URL https://github.com/enthought/mayavi.

[21] Dombi Szabolcs. ModernGL. 2016. URL https://github.com/cprogrammer1994/ModernGL.

[22] Holkner Alex. pyglet. 2007. URL https://bitbucket.org/pyglet/pyglet/wiki/Home.

[23] Stackoverflow. Random noise function for glsl. Online available from: https://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl.

[24] Green AD. A colour scheme for the display of astronomical intensity images. Bull Astron Soc India 2011;(39):289–95.

[25] Jaworski Michal. pyimgui. Online available from: https://github.com/swistakm/pyimgui.

[26] Rhiem Florian. glfw. Online available from: https://github.com/FlorianRhiem/pyGLFW.