FieldAnimation Documentation

Release 0.1.0

Nicola Creati, Roberto Vidmar

Sep 17, 2018

CONTENTS

CHAPTER

ONE

FOREWORD

FieldAnimation has been developed by two programmers working in the Aerial Operations group of the IRI Research Section at OGS - Istituto Nazionale di Oceanografia e di Geofisica Sperimentale.

Python is their favourite programming language since 2006.

The authors:

Nicola Creati, Roberto Vidmar

CHAPTER

TWO

WHAT IS FIELDANIMATION?

It is 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.

A background image can be shown to add information for the interpretation of the results.

An **example application** with interactive control of speed, color and number of animated particles is available in the examples direcory.

CHAPTER

THREE

FIELDANIMATION IN A NUTSHELL

- create a *fieldanimation.FieldAnimation* instance
- call its *fieldanimation.FieldAnimation.draw* method in the main rendering loop

CHAPTER

FOUR

DEPENDENCIES

It relies on PyOpenGL and numpy. The rendering of the OpenGL image must be carried out by a library that handles windows, input and events like GLFW or PyQt.

CHAPTER

FIVE

MORE

5.1 `fieldanimation` Package

fieldanimation classes:

`fieldanimation.`**`glInfo`**`()`
Return OpenGL information dict WARNING: OpenGL context MUST be initialized !!!

Parameters **`None`** –

Returns OpenGL information dict

`fieldanimation.`**`field2RGB`**(*field*)
Return 2D field converted to uint8 RGB image (i.e. scaled in [0, 255])

Parameters **`field`** (`numpy.ndarray`) – (u, v) 2D vector field instance

Returns

(rgb (`numpy.ndarray`): uint8 RGB image, uMin (float): u min, uMax (float): u max, vMin (float): v min, vMax (float): v max,) (tuple): Return value

`fieldanimation.`**`modulus`**(*field*)
Return normalized modulus of 2D field image

Returns normalized modulus of 2D field image (i.e. scaled in [0, 1.])

`class` `fieldanimation.`**`FieldAnimation`**(*width*, *height*, *field*, *computeSahder=False*, *image=None*)
Bases: `object`

Field Animation with OpenGL

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;

__init__(*width*, *height*, *field*, *computeSahder=False*, *image=None*)
Animate 2D vector field

Parameters

- **width** (*int*) – width in pixels
- **height** (*int*) – height in pixels
- **field** (*np.ndarray*) – 2D vector field
- **= True selects the compute shader version** (*cs*) –
- **= Optional background image** (*image*) –

setField(*field*)
Set the 2D vector field. Must be called every time a new vetor field is selected.

Parameters **field** (*np.ndarray*) – 2D vector field

setRenderingTarget(*texture*)
Set texture as rendering target

Parameters **(class** (texture) – *texture* instance): 2D vector field

setSize(*width*, *height*)
Set instance size. Must be called when the window is resized.

Parameters

- **width** (*int*) – window width in pixels
- **height** (*int*) – window height in pixels

resetRenderingTarget()
Bind first (default) framebuffer and reset the viewport.

tracersCount
Return tracers count

Returns number of tracers

_initTracers()
Initialize the tracers positions

draw()
Render the OpenGL scene. This method is called automatically when the scene has to be rendered and is responsible for the animation.

drawScreen()
Draw background texture and tracers on screen framebuffer texture

drawModulus(*opacity*)
Draw the modulus texture.

Parameters **opacity** (*float*) – opacity (alpha) of the texture: 0 –> transparent 1 –> opaque

drawImage()
Draw an image texture in background.

drawTexture(*texture*, *opacity*)
Draw *texture* on the screen.

Parameters

- **(** (texture) – class:Texture): texture instance
- **opacity** (*float*) – opacity (alpha) of the texture 0 –> transparent 1 –> opaque

drawTracers()
Draw the tracers on the screen

updateTracers()
Update tracers position using the fragment shader provided by the graphic card for computing.

updateTracersCS()
Update tracers position using the compute shader provided by the graphic card for computing.

5.1.1 shader Module

shader classes:

class fieldanimation.shader.**Shader**(*path=None*, ***kargs*)
Bases: object

Base shader class

__init__(*path=None*, ***kargs*)

addUniform(*uniform*, *utype=None*)
Add a uniform variable to the shader. If utype is not none it defines the setter function. Valid utypes are i, b, f, 2f, 4fv

addUniforms(*uniformlist*)

_build_shader(*sourceCode*, *shader_type*)
Actual building of the shader

delete()

_link()
Link the program

bind()
Bind the program, i.e. use it

unbind()
Unbind whatever program is currently bound - not necessarily this program, so this should probably be a class method instead.

setUniform(*name*, *value*)

setUniforms(*nameandvalue*)

code(*shader='v'*, *lineno=False*)
Return shader code.

Keyword Arguments

- **shader** (*string*) – v for vertex, f for fragment, g for geometry
- **lineno** (*boolean*) – add line numbers to code

Returns shader source code

vertex_code(*lineno=False*)
Return vertex shader code with optional line numbers

fragment_code(*lineno=False*)
Return fragment shader code with optional line numbers

geometry_code(*lineno=False*)
Return geometry shader code with optional line numbers

5.1.2 `texture` Module

texture classes:

class fieldanimation.texture.**Texture**(*data=None*, *width=None*, *height=None*, *filt=GL_NEAREST*, *dtype=GL_UNSIGNED_BYTE*)

Bases: object

__init__(*data=None*, *width=None*, *height=None*, *filt=GL_NEAREST*, *dtype=GL_UNSIGNED_BYTE*)
Texture object. If data is None an empty texture will be created

bind(*texUnit=0*)

handle()

unbind()

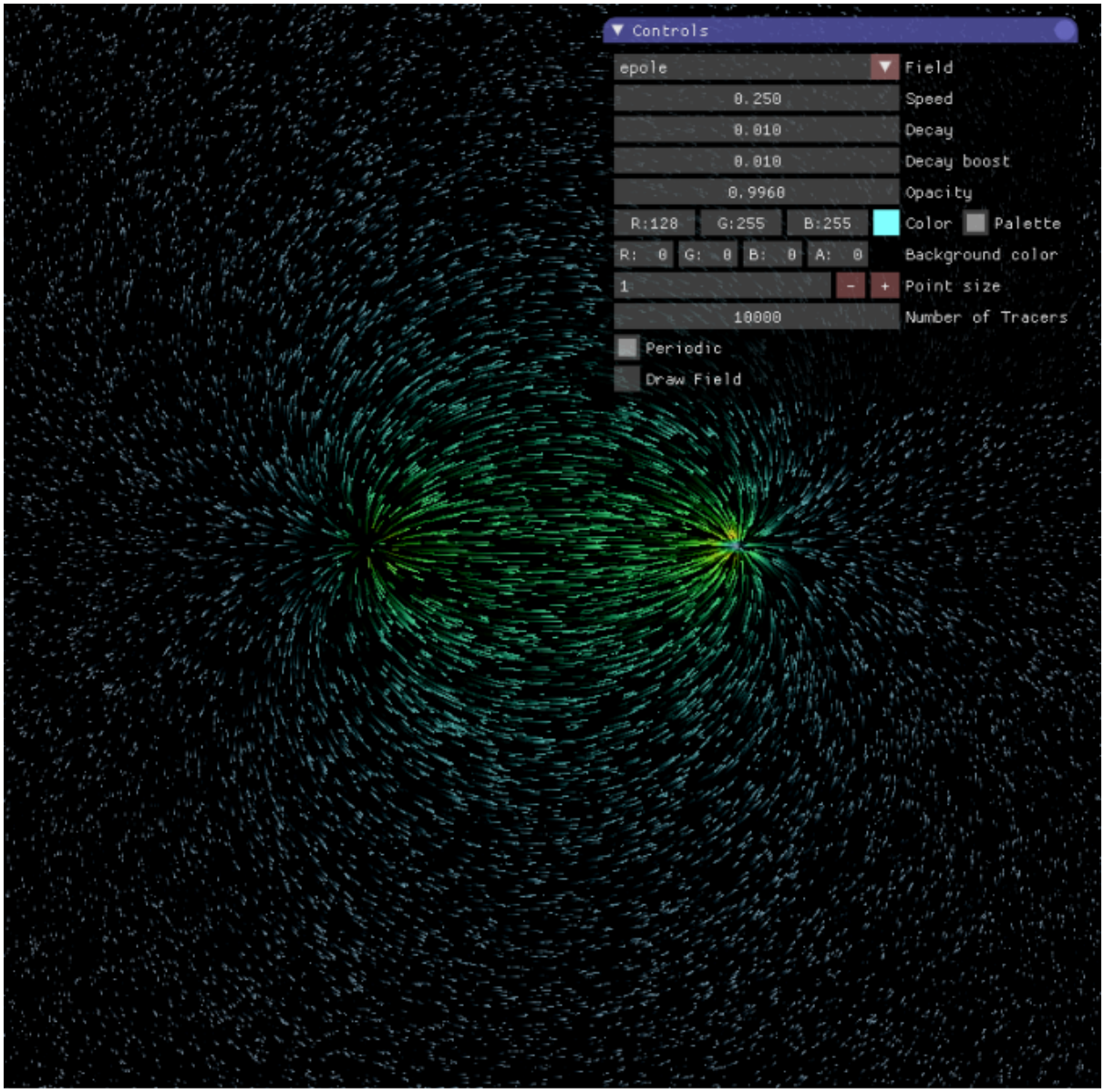
5.2 Example application

An example application appcode can be found in the examples directory.

The creation of a FieldAnimation image is straightforward: just instantiate the *fieldanimation.FieldAnimation* class passing the vector field array and call its draw method within the main rendering loop. The visualization application shown above depends on two OpenGL packages:

- pyimgui for setting interactively the visualization parameters and GLFW

for rendering the OpenGL image created by FieldAnimation in a windowing system.

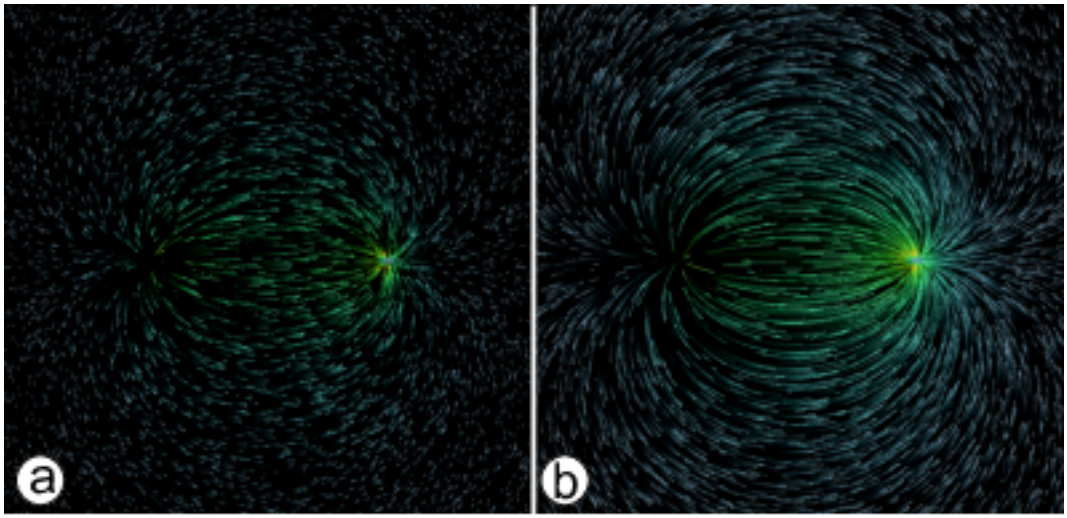


The interactive GUI in this figure allows to modify the visualization parameters that FieldAnimation embeds as instance attributes:

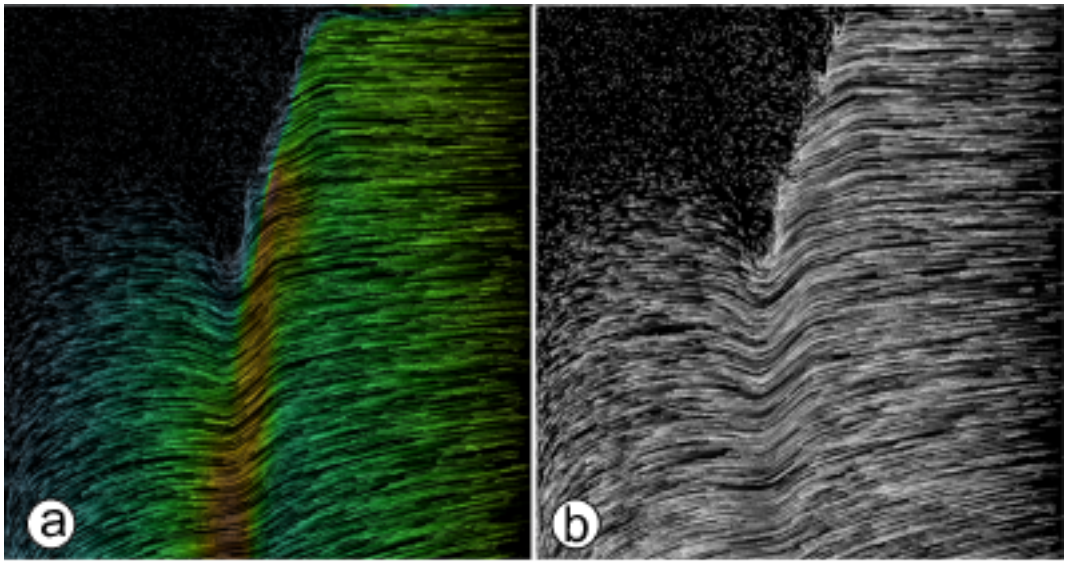
```
# Default values of the class attributes
FieldAnimation.speedFactor = 0.25
FieldAnimation.decay = 0.003
FieldAnimation.decayBoost = 0.01
FieldAnimation.fadeOpacity = 0.996
FieldAnimation.color = (0.5 , 1.0 , 1.0)
FieldAnimation.palette = True
FieldAnimation.pointSize = 1.0
FieldAnimation.tracersCount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

Here is a detailed description of the controls that appear in the GUI:

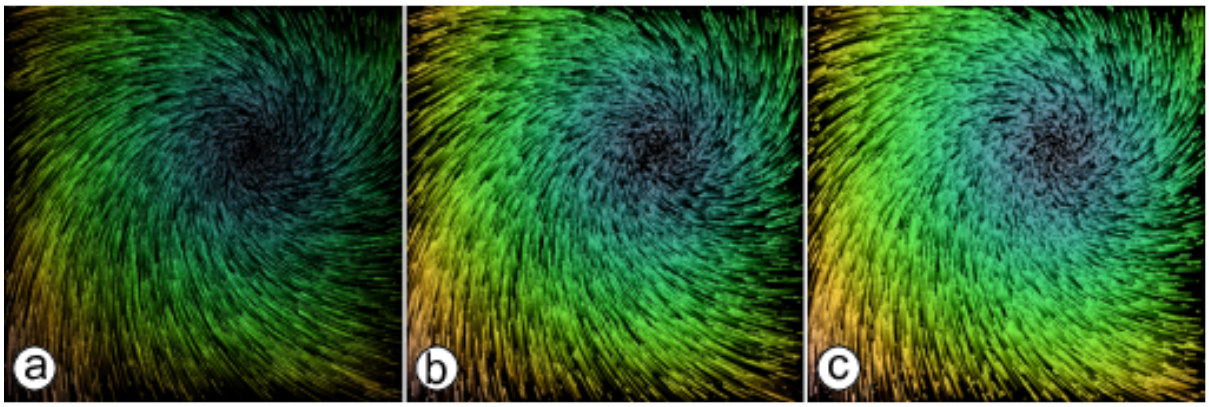
- **Field**: select one of the available vector fields implemented.
- **Speed**: set the length of the field pathlines i.e. the speed of the particles: the higher the value the longer the particle trace length.



- **Decay**: set the life span of a particle.
- **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 the field strength trough a cubehelix based color map.
- **Palette**: set color of the particles to a constant value.

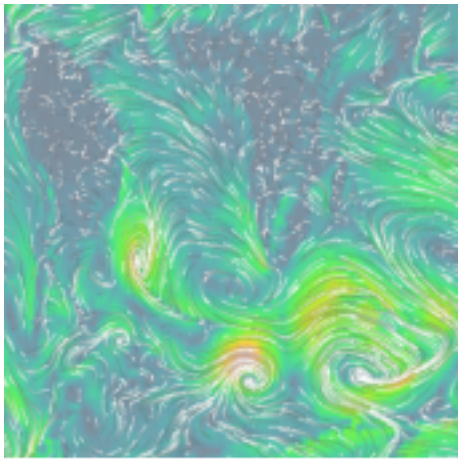


- **Point size**: set the size of the animated particles (in pixels).



- **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 this example application the field modulus is rendered trough a cubehelix color map.



5.3 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 or Paraview, 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 for fast visualization of a geodynamic deformation field. It implements a particles tracing visualization algorithm where particles move according to the field streamlines giving an instantaneous picture of its pattern and line flow. FieldAnimation has been designed to process bidimensional arrays and uses both PyOpenGL and numpy to take the most of the available hardware .

The adoption of the OpenGL programming pipeline 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. The programmer must only focus on the dataset and forget about the interfaces. The fieldanimation package consist of three modules: *fieldanimation*, *fieldanimation.texture* and *fieldanimation.shader*. The last two are just support modules that handle openGL textures and shader code loading, compiling and linking. *fieldanimation* is the core module in charge of arranging all the data, set the right parameters to the OpenGL context for rendering the scene and defines the drawing workflow. The animated image instance creation is straightforward:

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

- **width** and **height** are the vertical and horizontal pixels dimensions of the window
- **field** is an NxMx2 numpy array with the field components

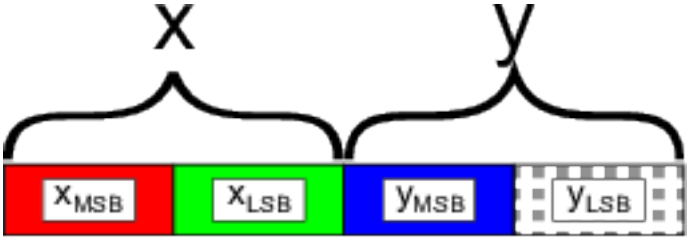
All modules are kept as simple as possible to reduce dependencies but new functionalities can be easily added subclassing *fieldanimation.FieldAnimation*. The package can be integrated in any windowing system.

5.3.1 Particle tracing algorithm

FieldAnimation 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 x 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



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 the original particles positions are retrieved from the RGBA texture using the "texture fetched method" in the vertex shader:

```
#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);
}
```

Decoding of particles position from texture is implemented through an array with absolute indexes of the particles, passed to the shaders.

Warning: This code has been tested *only* on Linux (Ubuntu 18.04.1 LTS and Ubuntu 17.10) but should work also on Mac and Windows.

Warning: This is work in progress!

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

F

G

H

M

R

S

T

U

V