Image Based Flow Visualization: Sample code

Supplemental material, ACM SIGGRAPH 2002

Jarke J. van Wijk

Technische Universiteit Eindhoven
Dept. of Mathematics and Computer Science

1 IBFV

In the article *Image Based Flow Visualization*, ACM SIGGRAPH 2002, we present a new method for the visualization of two-dimensional fluid flow. Figure 1 shows an example. This image illustrates three features of the method: handling of *unsteady flow*, *efficiency* and *ease of implementation*. More specifically, these 512×512 images are snapshots from an animation of a unsteady flow field. The animation was generated at 50 frames per second (fps) on a laptop computer. The method relies on rendering and blending texture mapped quadrilaterals. Modern graphics hardware, which can be found in standard PCs nowadays, is made exactly for this purpose, hence high speeds can be achieved.

To show the ease of implementation and portability of IBFV, the next page contains the complete C-source for the program that produced the animations of figure 1. This source code should enable the reader to make a quick start and to experiment by varying the parameters and extending visualization options. This version was written for compactness, rather than efficiency. For instance, the displaced mesh points are not stored separately and are calculated twice per frame.

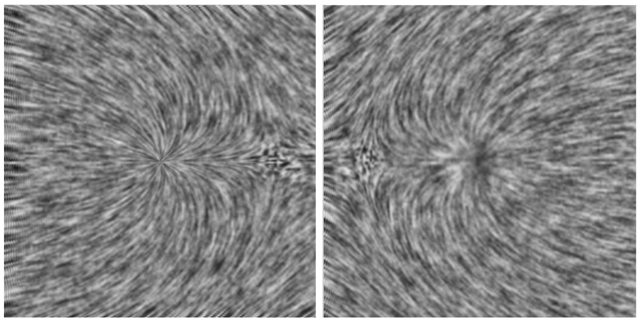

Figure 1: Texture visualization of flow

2 Description

In the article a detailed description of the method is given. Here we describe shortly how this method is translated into a sample program. For the user interface and the coupling with a window manager the well-known GLUT package was used. The meaning of the constants is as follows, with the symbol used in the article between parentheses:

`NPN`	Resolution of background pattern that produces fresh noise (N);
`NMESH`	Resolution of flow mesh (N_m);
`DM`	Distance in world coordinates between mesh lines;
`NPIX`	Resolution of image;
`SCALE`	Scale factor for background pattern (s). Use of a smaller value gives a more fine grained texture.

The global variables are:

`iframe`	frame counter (k);
`Npat`	Number of background patterns (M);
`alpha`	Blending factor (α);
`sa`	Strength of flow source (s_1);
`tmax`	Value of maximum texture coordinate;
`dmax`	Maximum displacement of texture ($v_{\max}\Delta t$).

In the main program GLUT is initialized first. Next, OpenGL is initialized in the function `initGL`. The world coordinates are set such that the lower left corner has coordinates [0, 0] and the upper right corner has coordinates [1, 1]. Furthermore, texture parameters are set, texturing mapping is enabled, the blending function is set, and the screen is cleared. The last part of the initialization is the generation of the noise patterns in the function `makePatterns`. This routine has to be called anew if properties of the patterns are changed interactively. A look-up table `lut` with 256 entries is used here to store the dynamic profile $w(t)$. In the first line this look-up table is filled with a simple block pulse. Next, for each pixel of the pattern a random phase is generated and stored in `phase` (ϕ_{ij} in the article). `Npat` patterns are generated in the next loops over the patterns and pixels. In the inner loop equation (18) is used. The time of the pattern plus a phase offset is used to determine the intensity of the pixel for the pattern. Furthermore, the alpha-blending value is set via the pattern, but this can be done in several other ways as well. The patterns are stored in the memory of the graphics card via display lists.

The function `display` produces a new frame. First, the strength of the flow source is set as a simple function of the frame number. Next, the distorted mesh is calculated and rendered. A rectangular mesh with equidistant mesh-lines is used. For each strip of the mesh a `GL_QUAD_STRIP` primitive is used. The coordinates `x1`, `x2` and `y1` are used as texture coordinates as well as world coordinates here. The texture coordinates are used to look up the intensity in the previous image, the world coordinates are used to get the distorted coordinates `px` and `py` via the function `getDP`. Obviously, this can be done in many other ways, using different meshes, different flow fields, and a more efficient calculation of the distorted mesh.

The flow field is defined by the function `getDP`. It returns the new position `(px, py)` of a point `(x, y)` in a flow field after a unit timestep. The function matches the description given in the article, see equation (26). Here we used a linear flow field $v_L = [0.02, 0]$ and a single flow element, with $\mathbf{p}_1 = [0.5, 0.5]$, $s_1 =$ `sa`, and $r_i = 0$. The maximum displacement is clamped to `dmax`, implicitly $\Delta t = 1$ is used.

After the previous step, a distorted version of the previous image is shown on the screen. Next, fresh noise is blended in. The appropriate noise pattern is selected, then one screen filling square is rendered and blended in, textured with the scaled noise pattern. Finally, the resulting image is copied directly to texture memory using `glCopyTexImage2D` (the only OpenGL1.1 call used here), ready for the next call of `display`.

Finally, we hope that the sample code will encourage the reader to apply and extend IBFV.

ibfv_sample.c

```
/*---------------------------------------------------------*/
/* ibfv_sample.c - Image Based Flow Visualization       */
/*                                                         */
/* Jarke J. van Wijk, 2002                                 */
/* Technische Universiteit Eindhoven                       */
/*---------------------------------------------------------*/
#include "GL/glut.h"
#include <stdlib.h>
#include <math.h>

#define NPN 64
#define NMESH  100
#define DM  ((float) (1.0/(NMESH-1.0)))
#define NPIX  512
#define SCALE 4.0

int    iframe = 0;
int    Npat   = 32;
int    alpha  = (0.12*255);
float  sa;
float  tmax   = NPIX/(SCALE*NPN);
float  dmax   = SCALE/NPIX;
/*---------------------------------------------------------*/
void initGL(void)
{
   glViewport(0, 0, (GLsizei) NPIX, (GLsizei) NPIX);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glTranslatef(-1.0, -1.0, 0.0);
   glScalef(2.0, 2.0, 1.0);
   glTexParameteri(GL_TEXTURE_2D,
                   GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D,
                   GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D,
                   GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D,
                   GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   glTexEnvf(GL_TEXTURE_ENV,
                   GL_TEXTURE_ENV_MODE, GL_REPLACE);
   glEnable(GL_TEXTURE_2D);
   glShadeModel(GL_FLAT);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glClear(GL_COLOR_BUFFER_BIT);
}
/*---------------------------------------------------------*/
void makePatterns(void)
{
   int lut[256];
   int phase[NPN][NPN];
   GLubyte pat[NPN][NPN][4];
   int i, j, k, t;

   for (i = 0; i < 256; i++) lut[i] = i < 127 ? 0 : 255;
   for (i = 0; i < NPN; i++)
   for (j = 0; j < NPN; j++) phase[i][j] = rand() % 256;

   for (k = 0; k < Npat; k++) {
      t = k*256/Npat;
      for (i = 0; i < NPN; i++)
      for (j = 0; j < NPN; j++) {
          pat[i][j][0] =
          pat[i][j][1] =
          pat[i][j][2] = lut[(t + phase[i][j]) % 256];
          pat[i][j][3] = alpha;
      }
      glNewList(k + 1, GL_COMPILE);
      glTexImage2D(GL_TEXTURE_2D, 0, 4, NPN, NPN, 0,
                   GL_RGBA, GL_UNSIGNED_BYTE, pat);
      glEndList();
   }
}
/*---------------------------------------------------------*/
void getDP(float x, float y, float *px, float *py)
{
   float dx, dy, vx, vy, r;

   dx = x - 0.5;
   dy = y - 0.5;
   r  = dx*dx + dy*dy;
   if (r < 0.0001) r = 0.0001;
   vx = sa*dx/r + 0.02;
   vy = sa*dy/r;
   r  = vx*vx + vy*vy;
   if (r > dmax*dmax) {
      r  = sqrt(r);
      vx *= dmax/r;
      vy *= dmax/r;
   }
   *px = x + vx;
   *py = y + vy;
}
/*---------------------------------------------------------*/
void display(void)
{
   int   i, j;
   float x1, x2, y, px, py;

   sa = 0.010*cos(iframe*2.0*M_PI/200.0);
   for (i = 0; i < NMESH-1; i++) {
      x1 = DM*i; x2 = x1 + DM;
      glBegin(GL_QUAD_STRIP);
      for (j = 0; j < NMESH; j++) {
          y = DM*j;
          glTexCoord2f(x1, y);
          getDP(x1, y, &px, &py);
          glVertex2f(px, py);

          glTexCoord2f(x2, y);
          getDP(x2, y, &px, &py);
          glVertex2f(px, py);
      }
      glEnd();
   }
   iframe = iframe + 1;

   glEnable(GL_BLEND);
   glCallList(iframe % Npat + 1);
   glBegin(GL_QUAD_STRIP);
      glTexCoord2f(0.0,  0.0);  glVertex2f(0.0, 0.0);
      glTexCoord2f(0.0,  tmax); glVertex2f(0.0, 1.0);
      glTexCoord2f(tmax, 0.0);  glVertex2f(1.0, 0.0);
      glTexCoord2f(tmax, tmax); glVertex2f(1.0, 1.0);
   glEnd();
   glDisable(GL_BLEND);
   glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
                    0, 0, NPIX, NPIX, 0);
   glutSwapBuffers();
}
/*---------------------------------------------------------*/
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(NPIX, NPIX);
   glutCreateWindow(argv[0]);
   glutDisplayFunc(display);
   glutIdleFunc(display);
   initGL();
   makePatterns();
   glutMainLoop();
   return 0;
}
```