Assignment #6
2D Vector Field Visualization—Arrow Plot and LIC
Due Oct.15th before midnight

Goal:
In this assignment, you will be asked to implement two visualization techniques for 2D steady
(time-independent) vector fields. The first technique is the direct visualization with colors and
arrows, the second one is a texture-based method -- line integral convolution (LIC). You will be
given a set of steady vector field data stored in PLY format for this assignment. All of them are
planar data and defined in a planar region [0,1]x[0,1].

Task:

1. Direct method (20 points)
1.1 Load the data as described in assignment #2.
You will need to modify your file loader because the data now contains only vector components and
does not contain scalar value. That means you have to remove variable “s” from
```
PlyProperty vert_props[] = { /* list of property information for a vertex */
    {"x", Float32, Float32, offsetof(Vertex_io,x), 0, 0, 0, 0},
    {"y", Float32, Float32, offsetof(Vertex_io,y), 0, 0, 0, 0},
    {"z", Float32, Float32, offsetof(Vertex_io,z), 0, 0, 0, 0},
    //{"s", Float32, Float32, offsetof(Vertex_io,s), 0, 0, 0, 0},
    {"vx", Float32, Float32, offsetof(Vertex_io,vx), 0, 0, 0, 0},
    {"vy", Float32, Float32, offsetof(Vertex_io,vy), 0, 0, 0, 0},
    {"vz", Float32, Float32, offsetof(Vertex_io,vz), 0, 0, 0, 0},
};
```
You should also modify the other part that is related to data loading (refer to the description of
assignment #2).

   In JS Version
You need to remove anything related to the `scalar: []` field that you added in the assignment 2 in the
plyLoader.js file.

1.2 Visualize the vector field magnitude, vector angle (0~2pi), x component, and y component,
respectively using color plots. That is, totally 4 different color plots. You should have the corresponding
interface to toggle on and off certain plots. (10 points)

   In JS Version
There are two shader programs in the skeleton code. You should use `shaderProgram` for the color plot
and leave `shaderTextureProgram` for the LIC texture in Part 2.

1.3 Visualize the vector field using arrow plots (10 points)
To create an arrow plot, draw an arrow at each vertex of the mesh.
Use the following routine to draw an arrow head with the given direction if you do not have one. You
should be able to draw the other part of the arrow that is just a line segment pointing from the vertex
location (x, y) to the direction according to the vector value defined at it. Note that you need to scale
the arrows uniformly through the whole field in order to get reasonable visualization.
void draw_arrow_head(double head[2], float direct[2])
{
    glPushMatrix();
    glTranslatef(head[0], head[1], 0);
    glRotatef(atan2(direct[1], direct[0])*360/(2*M_PI), 0, 0, 1);
    glScalef(0.03, 0.03, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f(0, 0);
    glVertex2f(-0.35, 0.12);
    glVertex2f(-0.35, -0.12);
    glEnd();
    glPopMatrix();
}

In JS Version
The corresponding function in Javascript is provided in the skeleton code.

/**
 * Draw an arrow
 * @param {*} x0 - The x coordinate value of the first point
 * @param {*} y0 - The y coordinate value of the first point
 * @param {*} x1 - The x coordinate value of the second point
 * @param {*} y1 - The y coordinate value of the second point
 * @param {*} modelViewMatrix
 * @param {*} projectionMatrix
 */
function draw_arrow(x0, y0, x1, y1, modelViewMatrix, projectionMatrix)

2. Line integral convolution (80 points)
Implement the LIC technique as described in the class. There are a number of different strategies to implement LIC. You are welcome to implement your own LIC algorithm. The following is my approach for your reference.

2.1 Set up three texture arrays
const int IMG_RES = 512;
unsigned char noise_tex[IMG_RES][IMG_RES][3];
unsigned char vec_img[IMG_RES][IMG_RES][3];
unsigned char LIC_tex[IMG_RES][IMG_RES][3];

In JS Version
var IMG_RES = 512;
var noise_tex = new Uint8Array(IMG_RES * IMG_RES * 3);
var vec_img = new Uint8Array(IMG_RES * IMG_RES * 4);
var LIC_tex = new Uint8Array(IMG_RES * IMG_RES * 4);
2.2 Create white noise texture
There are a number of different ways to generate a noise texture. For instance, you can use a 2D perlin noise as your input texture. The following use the random number generator provided by C/C++ to compute a noise texture.

```c
void gen_noise_tex ()
{
    for (int x = 0; x < IMG_RES; x++)
        for (int y = 0; y < IMG_RES; y++)
        {
            noise_tex[x][y][0] =
            noise_tex[x][y][1] =
            noise_tex[x][y][2] = (unsigned char) 255*(rand() % 32768) / 32768.0;
        }
}
```

**In JS Version**

```javascript
function gen_noise_tex() {
    var idx = 0;
    for (var x = 0; x < IMG_RES; x++)
        for (var y = 0; y < IMG_RES; y++) {
            noise_tex[idx] =
            noise_tex[idx + 1] =
            idx = idx + 3;
        }
}
```

2.3 Encode the 2D vector field into an image
This is for the simplicity of the later streamline computation. We encode the 2D vector field into an image with the exactly same resolution of the output image. This image will be created once for each field that is loaded.

**First**, assume the vector value is in the form of (vx, vy) at each vertex. Then, search the maximum and minimum x and y components of the vector values through the entire field, i.e. max_vx, min_vx, max_vy, and min_vy.

**Second**, render the vector field into an image such that the image encodes both the x and y components. Specifically, for a vector value (vx, vy) at a vertex, we compute the color at each mesh vertex as follows

```c
float rgb[3];
rgb[0] = (vx - min_vx) / (max_vx - min_vx);   // red channel
rgb[1] = (vy - min_vy) / (maxVy - minVy);    // green channel
rgb[2] = 0;
```

**In JS Version**

```javascript
var rgb = [];
```
rgb[0] = (vx - min_vx) / (max_vx - min_vx);  // red channel
rgb[1] = (vy - min_vy) / (max_vy - min_vy);  // green channel
rgb[2] = 0;

You can create a routine as follows and call it right after you load the data and set up the initialization. Remember the LIC image need to compute only once and will be saved as a texture for later rendering.

```c
void render_vec_img( Polyhedron *this_poly)
{
    glViewport(0, 0, (GLsizei) 512, (GLsizei) 512);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 1, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawBuffer(GL_BACK);
    int i, j;
    // first search the max_vx, min_vx, max_vy, min_vy through the entire field ...
    // render the mesh
    for (i=0; i<this_poly->ntris; i++) {
        Triangle *temp_t=this_poly->tlist[i];
        float rgb[3];
        rgb[2] = 0.5;
        glBegin(GL_TRIANGLES);
        for (j=0; j<3; j++)
        {
            Vertex *v = temp_t->verts[j];
            //determine the color for this vertex based on its vector value ...
            glVertex2f (v->x, v->y);
        }
        glEnd();
    }
    // save the rendered image into the vec_img
    glReadBuffer(GL_BACK);
    glReadPixels(0, 0, 512, 512, GL_RGB, GL_UNSIGNED_BYTE, vec_img);
}
```

In JS Version
```
function render_vec_img() {
    canvas.width = IMG_RES;
    canvas.height = IMG_RES;
    gl.viewport(0, 0, IMG_RES, IMG_RES);
    gl.clearColor(0.0, 0.0, 0.0, 1.0);  // Clear to black, fully opaque
    gl.clearDepth(1.0);                 // Clear everything
    gl.enable(gl.DEPTH_TEST);           // Enable depth testing
    gl.depthFunc(gl.LEQUAL);            // Near things obscure far things

    // Clear the canvas before we start drawing on it.
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```
gl clearColor(0.0, 0.0, 0.0, 1.0);  // Clear to black, fully opaque
gl clearDepth(1.0);                 // Clear everything
gl enable(gl.DEPTH_TEST);           // Enable depth testing
gl depthFunc(gl.LEQUAL);            // Near things obscure far things

// Clear the canvas before we start drawing on it.

gl clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

const projectionMatrix = mat4.create();

//Generates a orthogonal projection matrix with the given bounds
console.log(min_x, max_x, min_y, max_y);
const left = min_x;
const right = max_x;
const bottom = min_y;
const top = max_y;
const zNear = 0.1;
const zFar = 100;
mat4.ortho(projectionMatrix,
    left,
    right,
    bottom,
    top,
    zNear,
    zFar);

// Set the drawing position to the "identity" point, which is
// the center of the scene.
const modelViewMatrix = mat4.create();
// Update the model view matrix if there are some changes in translation and
rotation
mat4.translate(modelViewMatrix,     // destination matrix
    modelViewMatrix,     // matrix to translate
    [0, 0., -1]);  // amount to translate

mat4.rotate(modelViewMatrix,     // destination matrix
    modelViewMatrix,     // matrix to rotate
    transform.angleZ,     // amount to rotate in radians
    [0, 0, 1]);       // axis to rotate around (Z)

mat4.rotate(modelViewMatrix,     // destination matrix
    modelViewMatrix,     // matrix to rotate
    transform.angleY, // amount to rotate in radians
    [0, 1, 0]);       // axis to rotate around (Y)

mat4.rotate(modelViewMatrix,     // destination matrix
modelViewMatrix,  // matrix to rotate
transform.angleX, // amount to rotate in radians
[1, 0, 0]);

// first search the max_vx, min_vx, maxVy, minVy through the entire field...

// determine the color for this vertex based on its vector value...

// draw the mesh...

var tmp_vec_img = new Uint8Array(IMG_RES * IMG_RES * 4);
gl.readPixels(0, 0, IMG_RES, IMG_RES, gl.RGBA, gl.UNSIGNED_BYTE, tmp_vec_img);

// set values for the vec_img. In comparison with C++, gl.readPixels in WebGL inverses row and column.

2.4 Compute LIC image

The basic framework for computing a LIC image is as follows.

For each pixel

  // Compute a streamline using the vector field image in forward and backward direction (the streamline computation is terminated when the desired number of pixels is reached)
  // Accumulate the color values from the pixels obtained in the previous step

As you have encoded the vector field into an image with the same resolution of the output in the previous step. The streamline tracing and the search of the pixels that it passes can be combined. Here is the basic idea of how these two can be combined.

Given a current pixel (i, j) with i, j being its integer indexes corresponding to its row and column positions, respectively.

We start from the center of this pixel

  float y = i+.5;
  float x = j+.5;

Let

  int next_i = i;
  int next_j = j;

and extract the vector value from the vec_img as follows:

  vx = min_vx + (max_vx - min_vx) * vec_img[next_i][next_j][0]/255.0;
  vy = min_vy + (max_vy - min_vy) * vec_img[next_i][next_j][1]/255.0;

  In JS Version
  vx = min_vx + (max_vx - min_vx) * vec_img[(next_i+ next_j* IMG_RES ) * 4] / 255.0;
\[ vy = \min_{vy} + (\max_{vy} - \min_{vy}) \times \text{vec}_img[(\text{next}_i + \text{next}_j \times \text{IMG}_RES) \times 4 + 1] / 255.0; \]

This vector \((vx, vy)\) is normalized to be a unit vector so that you can move to one of its eight neighboring pixels. The next position for the streamline computation can then be estimated using Euler integration.

\[
x = x + vx;
y = y + vy;
\]

Then,

\[
\text{next}_i = \text{int}(x);
\text{next}_j = \text{int}(y);
\]

is the next pixel the streamline will pass. Save it into an array.

We repeat the above process until
1) we hit the boundary of the domain, i.e. next_i<0 or next_j<0 or next_i>=\text{IMG}_RES or next_j>=\text{IMG}_RES,
or 2) the current vector value is smaller than some threshold, i.e. reaching a fixed point, or
3) the total number of pixels that we have visited is \(L/2\) (Assume \(L\) is the kernel size).

The backward tracing can be similarly implemented. 

NOTE that this is a really coarse estimation of the streamline and will introduce large error. You can consider to resort to high-ordered integrators to improve the accuracy.

Then the color of the current pixel can be computed as the weighted sum of the color values of those pixels extracted in the streamline tracing. In this assignment, a simple average of those colors is sufficient.

### 2.5 Render the LIC image through texture mapping

Replace your `Display()` function with the following one since we are working on 2D vector field now.

```c
void Display()
{
    glViewport(0, 0, (GLsizei) IMG_RES, (GLsizei) IMG_RES);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 1, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    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);

    // Test noise texture (for debugging purpose)
    // glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, IMG_RES, IMG_RES, 0,
    //    GL_RGB, GL_UNSIGNED_BYTE, noise_tex);

    // Test vector field image (for debugging purpose)
    // glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, IMG_RES, IMG_RES, 0,
```

// GL_RGB, GL_UNSIGNED_BYTE, vec_img);

// Display LIC image using texture mapping
GLuint LIC_tex;

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, IMG_RES, IMG_RES, 0,
               GL_RGB, GL_UNSIGNED_BYTE, LIC_tex);

glBegin(GL_QUAD_STRIP);
glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
glTexCoord2f(0.0, 1.0); glVertex2f(0.0, 1.0);
glTexCoord2f(1.0, 0.0); glVertex2f(1.0, 0.0);
glTexCoord2f(1.0, 1.0); glVertex2f(1.0, 1.0);

glEnd();

glDisable(GL_TEXTURE_2D);

// Add you arrow plot here and use a checkbox to enable its visualization

// If you are using AntTweakBar, please call the following to draw it

TwDraw();

glutSwapBuffers();
glFlush();

}
Generates a orthogonal projection matrix with the given bounds
console.log(min_x, max_x, min_y, max_y);
const left = min_x;
const right = max_x;
const bottom = min_y;
const top = max_y;
const zNear = 0.1;
const zFar = 100;
mat4.ortho(projectionMatrix, 
    left,
    right,
    bottom,
    top,
    zNear,
    zFar);

// Set the drawing position to the "identity" point, which is 
// the center of the scene.
const modelViewMatrix = mat4.create();
// Update the model view matrix if there are some changes in translation and 
rotation
mat4.translate(modelViewMatrix,     // destination matrix
    modelViewMatrix,     // matrix to translate
    [0, 0., -1]);  // amount to translate

mat4.rotate(modelViewMatrix, // destination matrix
    modelViewMatrix, // matrix to rotate
    transform.angleZ, // amount to rotate in radians
    [0, 0, 1]);       // axis to rotate around (Z)

mat4.rotate(modelViewMatrix, // destination matrix
    modelViewMatrix, // matrix to rotate
    transform.angleY,// amount to rotate in radians
    [0, 1, 0]);       // axis to rotate around (Y)

mat4.rotate(modelViewMatrix, // destination matrix
    modelViewMatrix, // matrix to rotate
    transform.angleX,// amount to rotate in radians
    [1, 0, 0]);

// define size and format of level 0
const level = 0;
const internalFormat = gl.RGBA;
const border = 0;
const format = gl.RGBA;
const type = gl.UNSIGNED_BYTE;

const targetTexture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, targetTexture);
```
gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, IMG_RES, IMG_RES, border, format,  
    type, LIC_tex);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
gl.texParameterf(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.pixelStorei(gl.UNPACK_ALIGNMENT, 1);

gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.enable(gl.BLEND);

var textureCoordinates = [
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0,
    0.0, 1.0,
];

var positions = [
    min_x, min_y, 0,
    min_x, max_y, 0,
    max_x, max_y, 0,
    max_x, min_y, 0,
];

var indices = [
    0, 1, 2,
    0, 2, 3,
];
const buffers = initBuffers(positions, textureCoordinates, indices);

// Use draw_texture_buffers(targetTexture, buffers, modelViewMatrix,  
// projectionMatrix) to draw LIC texture

    // Draw arrows here

}

You can also add a checkbox to your interface to toggle on and off this rendering mode.

If you use AntTweakBar, please modify your Reshape() function as follows

void Reshape(int width, int height)
{
    // Set OpenGL viewport and camera
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 1, 0, 1);
    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```
3. Enhance the original LIC (extra)
You can enhance the obtained LIC result from the previous task by doing the following.

3.1 Visualize vector magnitude with the LIC image (5 points)
You can use OpenGL blending function to blend the color plot showing the vector magnitude with the LIC image. Another option is to vary the kernel length (e.g. L) for each pixel according to the magnitude of the vector valued defined at the current pixel. Try to explore that by yourself.

3.2 Enhanced LIC (5 points)
Implement a simplified version of Enhanced LIC. After getting the initial LIC in Task 2, use it as the input (i.e. replace the original white noise) to perform one more LIC process.

Grades:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Total Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>10 (extra)</td>
</tr>
</tbody>
</table>

Here are the results for the five data sets you should expect to see.