AN OPEN-SOURCE, HARDWARE ACCELERATED INDUSTRY PROTOTYPE FOR MULTIDIMENSIONAL GLYPHS – SUPPLEMENTARY MATERIAL

Dylan Rees
Swansea University
Swansea, UK

Robert S. Laramee
University of Nottingham
Nottingham, UK

1. INTRODUCTION

This article provides supplementary material to An Open-Source, Hardware Accelerated Industry Prototype for Multidimensional Glyphs article. The main article introduces a technique for utilizing the OpenGL rendering pipeline to determine the geometry of, and render complex data dependent glyphs, exploiting the parallel processing power of a Graphical Processing Unit. This supplementary material provides details of an open-source implementation of the technique to provide reproducibility. Source code is also available from an online repository on GitHub:

https://github.com/glyph-renderer/glyph

The source code has been written for compactness and therefore contains no tests for the presence or correctness of the OpenGL shaders or libraries. The code as presented requires OpenGL 4.6. Once the code is running, the glyph design can be changed by pressing the `C` key for circular glyphs, and the `B` key for bar chart glyphs. The `Escape` key terminates the application.

2. PERFORMANCE RESULTS

To gauge the performance of the presented technique, the number of frames rendered each second was measured for a varying number of glyphs. Results are shown in Table 1.

Table 1: Frame time rendering performance for drawing differing numbers of glyphs

Number of Glyphs	**Frame Rendering Time (ms)**					
	Bar Glyph (With Commodity Graphics Card)	Circular Glyph (With Commodity Graphics Card)	Star Glyph (With Commodity Graphics Card)	Bar Glyph (Without Graphics Card)	Circular Glyph (Without Graphics Card)	Star Glyph (Without Commodity Graphics Card)
1,000	<1	<1	<1	2.0	7.6	1.9
2,500	<1	2.0	<1	3.9	13	3.6
5,000	1.2	4.0	<1	6.9	34	6.4
7,500	1.7	5.6	1.3	9.9	53	9.1
10,000	2.2	7.3	1.7	13	58	12
20,000	4.2	14	3.1	25	113	22
30,000	6.0	21	4.5	44	140	40
40,000	8.0	28	6.0	58	188	52
50,000	10	35	7.4	60	235	54
60,000	12	42	9.0	72	284	65

70,000	14	48	10	84	330	76
80,000	15	55	12	95	377	87
90,000	17	61	13	107	424	97
100,000	20	71	14	119	471	108
110,000	21	77	16	131	517	118
120,000	23	82	17	143	564	130
130,000	25	91	19	155	611	140
140,000	27	98	20	167	685	151
150,000	29	105	21	178	705	162
160,000	30	109	23	190	752	172
180,000	25	129	26	214	854	194
200,000	39	141	29	238	939	216
220,000	42	155	31	262	1030	237

3. OPEN SOURCE CODE DESCRIPTION

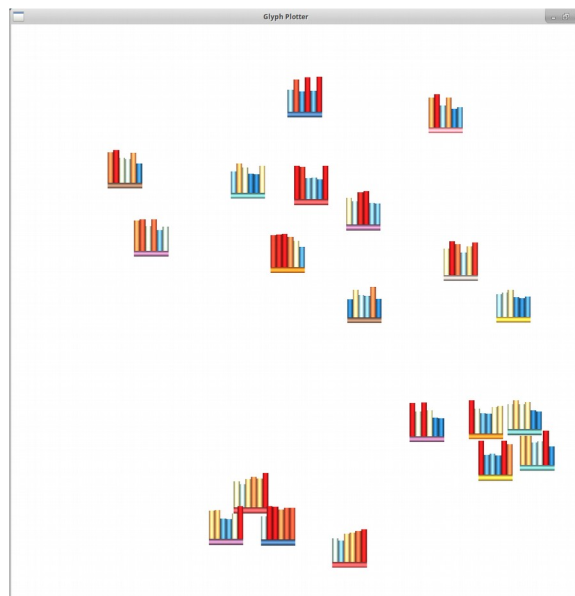

Figure 1: A screenshot of the open-source implementation featuring 20 randomly generated bar glyphs

Here we detail how the technique is presented in a simple program pictured in Figure 1. Four libraries are utilized to provide functions for the operation of the program. The *GLEW* library provides access to OpenGL functions, the *GLFW* library provides the user interface, window, and key interaction functions, file reading functions are provided from the *stdio* library, and the *stdlib* library provides a random number generator for data creation. All libraries provide cross platform support.

The defined constants are as follows:

- **`NPIX`** - Resolution of the OpenGL canvas.
- **`MAX_SZ`** -The maximum size of a shader script.
- **`NO_GLYPHS`** - The number of glyphs to generate and render.
-

The global variables are:

- **`proj_matrix[] -`** A 4 by 4 matrix establishing the OpenGL viewport projection. The matrix establishes the lower left corner of the canvas as [-0.1, -0.1] and the top right corner [1.1, 1.1].
- **`cat_colormap[]`** - The discrete colormap for the category indicator. Colors are provided as RGB unsigned integers between 0-255.

- **cat_map_sz** The number of color variables in cat_colormap[].
- **data_colormap[] -** The continuous colormap for data mapping. Colors are provided as RGB unsigned integers between 0-255.
- **data_map_sz** - The number of color variables in data_colormap[].
- **pos_sz -** The number of position coordinates for a single instance.

The main program starts by initializing GLFW and then creating the interface window. Following this, eight arrays are initiated for data storage. The first array, points, is for storing glyph positions, with two figures for each glyph representing the x and y coordinates. The second array, cols, stores categorical values for each glyph. The final six arrays store data variables that determine the glyph geometry.
A for loop is then used to populate the category glyph with integers between 1-10, and the position array with both x and y values 0-1.

A Vertex Array Object is created and bound ready for the addition of the first Buffer Array Object which is created and populated with the points array. A second Buffer Array Object is then created and populated with the cols array. Following this, Buffer Array Objects are created and the randomArray function is used to generate random data to populate the data variable arrays and to add each to a Buffer Array Object.

The draw function is then called. This function is used to compile the OpenGL program and render scenes. Within the draw function, the loadShader function takes three GLSL shader file names as an input. This function then reads in each shader file in turn and attaches them to compile a shader program, assigns the projection matrix to the program, and creates and attaches textures to the program.

Once a shader program is created, a rendering while loop is entered. Within the while loop, the canvas is cleared and set to a white background. The glDrawArrays function is then used to draw the scene. Following this, *GLFW* functions are used to check for keyboard inputs before the loop repeats. If a B or C or S key is pressed, the realoadShader function recompiles the shader with bar, circular or star glyph, respectively. The ESCAPE key causes the program to close.

glyphRenderer.cpp

```
/*----------------------------------------------------*/
/* glyphRenderer.cpp -Multidimensional Glyph Rendering*/
/*                                                    */
/*                                                    */
/*----------------------------------------------------*/
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stdio.h>
#include <stdlib.h>

#define NPIX  1025
#define MAX_SZ 100000
#define NO_GLYPHS 20

GLfloat proj_matrix[] = {
                    1.667f, 0.0f, 0.0f, 0.0f,
                    0.0f, 1.667f, 0.0f, 0.0f,
                    0.0f, 0.0f, 1.0f, 0.0f,
                    -0.833f, -0.833f, 0.0f, 1.0f
          };
GLubyte cat_colormap[] = {
                    78,121,167, 242,142,44, 225,87,89,
                    118,183,178, 89,161,79, 237,201,73,
                    175,122,161, 255,157,167,
                    156,117,95, 186,176,171
          };
int       cat_map_sz = 10;
GLubyte data_colormap[] = {
                    44,123,182, 171,217,233,
                    55,255,191, 253,174,97, 215,25,28
          };
int       data_map_sz = 5;
int       pos_sz = 2;
/*----------------------------------------------------*/
GLuint makeShader( const char* shader_filename,
                   GLenum shaderType ) {
  char sdr_str[MAX_SZ];
  FILE* sdr_file = fopen( shader_filename, "r" );
  size_t shdr_sz = fread( sdr_str, 1,
                          MAX_SZ - 1, sdr_file );
  sdr_str[shdr_sz] = '\0';
  fclose( sdr_file );

  GLuint shader_handle   = glCreateShader( shaderType );
  const GLchar *shader_ptr = (const GLchar *)sdr_str;
  glShaderSource( shader_handle, 1, &shader_ptr, NULL );
  glCompileShader( shader_handle );

  return shader_handle;
}
/*----------------------------------------------------*/
GLuint loadShader(const char* vert_shad_filen,
                      const char* frag_shad_filen,
                      const char* geom_shad_filen ) {

  GLuint shader_program = glCreateProgram();
  GLuint vert_shad_handle = makeShader(vert_shad_filen,
                                       GL_VERTEX_SHADER);
  GLuint frag_shad_handle = makeShader(frag_shad_filen,
                                     GL_FRAGMENT_SHADER);
  GLuint geom_shad_handle = makeShader(geom_shad_filen,
                                     GL_GEOMETRY_SHADER);

  glAttachShader( shader_program, vert_shad_handle );
  glAttachShader( shader_program, frag_shad_handle );
  glAttachShader( shader_program, geom_shad_handle );

  glLinkProgram( shader_program );
  glDeleteShader( vert_shad_handle );
  glDeleteShader( frag_shad_handle );
  glDeleteShader( geom_shad_handle );

  glLinkProgram(shader_program);
  int matrix_location = glGetUniformLocation(
                            shader_program,"matrix");
  int cat_tex_location = glGetUniformLocation(
                       shader_program, "cat_texture");
  int attr_tex_location = glGetUniformLocation(
                      shader_program, "attr_texture");
  glUseProgram( shader_program );
  glUniformMatrix4fv( matrix_location, 1, GL_FALSE,
                      proj_matrix );
  glUniform1i(cat_tex_location, 0);
  glUniform1i(attr_tex_location,  1);

  GLuint cat_textureID;
  glGenTextures(1, &cat_textureID);
  glActiveTexture(GL_TEXTURE0 + 0);
  glBindTexture(GL_TEXTURE_1D, cat_textureID);
  glTexImage1D( GL_TEXTURE_1D, 0, GL_RGB, cat_map_sz, 0,
             GL_RGB, GL_UNSIGNED_BYTE, cat_colormap );
  glGenerateMipmap( GL_TEXTURE_1D );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S,
                   GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T,
                   GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST );

  GLuint attr_textureID;
  glGenTextures(1, &attr_textureID);
  glActiveTexture(GL_TEXTURE0 + 1);
  glBindTexture(GL_TEXTURE_1D, attr_textureID);
  glTexImage1D( GL_TEXTURE_1D, 0, GL_RGB, data_map_sz,
          0, GL_RGB, GL_UNSIGNED_BYTE, data_colormap );
  glGenerateMipmap( GL_TEXTURE_1D );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S,
                   GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T,
                   GL_CLAMP_TO_EDGE );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR );
  glTexParameteri( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR );

  return shader_program;
}
/*-------------------------------------------------------
*/
void reloadShader( GLuint* program,
                   const char* vert_shad_filen,
                   const char* frag_shad_filen,
                   const char* geom_shad_filen ) {

  GLuint reloaded_program = loadShader( vert_shad_filen,
                                        frag_shad_filen,
                                       geom_shad_filen );
  if ( reloaded_program ) {
      glDeleteProgram( *program );
      *program = reloaded_program;
  }
}
/*----------------------------------------------------*/
void randomArray( float* array, GLuint* buffer,
                  int attrib_pos) {

  for (int i=0;i<NO_GLYPHS;i++) {
          array[i] = float(rand()) / float(RAND_MAX);
  }
  glBindBuffer(GL_ARRAY_BUFFER, *buffer);
  glBufferData(GL_ARRAY_BUFFER, NO_GLYPHS
          *sizeof(float), array, GL_STATIC_DRAW);
  glVertexAttribPointer( attrib_pos, 1, GL_FLOAT,
                         GL_FALSE, 0, NULL);
  glEnableVertexAttribArray(attrib_pos);
}
/*----------------------------------------------------*/
void draw(GLFWwindow* window, GLuint vao) {

  GLuint shader_program = loadShader( "vert.glsl",
                                      "frag.glsl",
                                      "geom.glsl" );

  while(!glfwWindowShouldClose(window)) {
          glClear(GL_COLOR_BUFFER_BIT |
                                   GL_DEPTH_BUFFER_BIT);
          glClearColor( 1.0f, 1.0f, 1.0f, 1.0f);
          glUseProgram(shader_program);
          glBindVertexArray(vao);
          glDrawArrays(GL_POINTS, 0, NO_GLYPHS);
          glfwPollEvents();
          glfwSwapBuffers(window);
          if ( glfwGetKey( window, GLFW_KEY_B ) ) {
                    reloadShader( &shader_program,
                                  "vert.glsl",
                                  "frag.glsl",
                                  "geom.glsl" );
          }
          if ( glfwGetKey( window, GLFW_KEY_C ) ) {
                    reloadShader( &shader_program,
                                  "vert.glsl",
                                  "frag.glsl",
                                  "geomAlt.glsl" );
          }
          if ( glfwGetKey( window, GLFW_KEY_S ) ) {
                    reloadShader( &shader_program,
                                  "vert.glsl",
                                  "frag.glsl",
                                  "geomStar.glsl" );
          }
```

```
          if(glfwGetKey(window, GLFW_KEY_ESCAPE)) {
                    glfwSetWindowShouldClose(window, 1);
          }
  }
}
/*---------------------------------------------------*/
int main() {

  glfwInit();
  GLFWwindow* window = glfwCreateWindow(NPIX, NPIX,
                                        "Glyph Plotter",
                                        NULL, NULL);
  glfwMakeContextCurrent( window );
  glewExperimental = GL_TRUE;
  glewInit();

  float points[pos_sz*NO_GLYPHS];
  float cols[NO_GLYPHS];
  float varA[NO_GLYPHS];
  float varB[NO_GLYPHS];
  float varC[NO_GLYPHS];
  float varD[NO_GLYPHS];
  float varE[NO_GLYPHS];
  float varF[NO_GLYPHS];
  for (int i=0;i<NO_GLYPHS;i++) {
          cols[i] = rand() % 10 + 1;
          points[i*pos_sz] =
float(rand())/float(RAND_MAX);
          points[i*pos_sz+1] =
float(rand())/float(RAND_MAX);
  }
  GLuint vao = 0;
  glGenVertexArrays(1, &vao);
  glBindVertexArray(vao);

  GLuint points_vbo = 0;
  glGenBuffers(1, &points_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
  glBufferData(GL_ARRAY_BUFFER, pos_sz* NO_GLYPHS *
                 sizeof(float), points, GL_STATIC_DRAW);
  glVertexAttribPointer(0, pos_sz, GL_FLOAT,
                        GL_FALSE, 0, NULL);
  glEnableVertexAttribArray(0);

  GLuint colours_vbo = 0;
  glGenBuffers(1, &colours_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, colours_vbo);
  glBufferData(GL_ARRAY_BUFFER,
               NO_GLYPHS * sizeof(float),
               cols, GL_STATIC_DRAW);
  glVertexAttribPointer(1, 1, GL_FLOAT,
                        GL_FALSE, 0, NULL);
  glEnableVertexAttribArray(1);

  GLuint varA_vbo = 0;
  glGenBuffers(1, &varA_vbo);
  randomArray(varA, &varA_vbo,2);
  GLuint varB_vbo = 0;
  glGenBuffers(1, &varB_vbo);
  randomArray(varB, &varB_vbo,3);
  GLuint varC_vbo = 0;
  glGenBuffers(1, &varC_vbo);
  randomArray(varC, &varC_vbo,4);
  GLuint varD_vbo = 0;
  glGenBuffers(1, &varD_vbo);
  randomArray(varD, &varD_vbo,5);
  GLuint varE_vbo = 0;
  glGenBuffers(1, &varE_vbo);
  randomArray(varE, &varE_vbo,6);
  GLuint varF_vbo = 0;
  glGenBuffers(1, &varF_vbo);
  randomArray(varF, &varF_vbo,7);

  draw(window, vao);
  glfwTerminate();
  return 0;
}
```

vert.glsl

```
#version 460

layout(location = 0) in vec2 vertex_pos;
layout(location = 1) in float cat_color;
layout(location = 2) in float barAattr;
layout(location = 3) in float barBattr;
layout(location = 4) in float barCattr;
layout(location = 5) in float barDattr;
layout(location = 6) in float barEattr;
layout(location = 7) in float barFattr;
out float barA;
out float barB;
out float barC;
out float barD;
out float barE;
out float barF;
out float color;

void main() {
    gl_Position = vec4(vertex_pos, 0.0f, 1.0f);
    barA = barAattr;
    barB = barBattr;
    barC = barCattr;
    barD = barDattr;
    barE = barEattr;
    barF = barFattr;
    color = cat_color;
}
```

geom.glsl

```
#version 460


uniform mat4 matrix;
layout(points) in;
in float barA[];
in float barB[];
in float barC[];
in float barD[];
in float barE[];
in float barF[];
in float color[];
layout (triangle_strip, max_vertices = 200) out;
out float barColor;
out float pass;
out vec2 position;

float glyphSize = 0.06f;
float barWidth = glyphSize/3.0;
float noOfVariables = 6.0;
float halfVar = noOfVariables/2.0;

void main() {
    vec4 center = gl_in[0].gl_Position;
    vec4 pos = matrix*center;

    //draw variable bars
    pass = 0;
    for (int i=0; i<noOfVariables;i++) {
        float barValue = 0.0f;
        switch (i) {
        case 0: barValue = barA[0]; break;
        case 1: barValue = barB[0]; break;
        case 2: barValue = barC[0]; break;
        case 3: barValue = barD[0]; break;
        case 4: barValue = barE[0]; break;
        case 5: barValue = barF[0]; break;
        default: barValue = 0.0f;  break;
        }
        float barMultiply= barValue*glyphSize+glyphSize;
        barColor = barValue;

        float xPosAdj = float(i)-halfVar;
        gl_Position = ( pos + vec4(xPosAdj*barWidth,
                          0.0-glyphSize, 0.0, 0.0));
        position =vec2(0.f,0.f);
        EmitVertex();
        gl_Position = ( pos + vec4((xPosAdj+1)*barWidth,
                          0.0-glyphSize, 0.0, 0.0));
        position =vec2(1.f,0.f);
        EmitVertex();
        gl_Position = ( pos + vec4(xPosAdj*barWidth,
                      barMultiply-glyphSize, 0.0, 0.0));
        position =vec2(0.f,1.f);
        EmitVertex();
        gl_Position = ( pos + vec4((xPosAdj+1)*barWidth,
                      barMultiply-glyphSize, 0.0, 0.0));
        position =vec2(1.f,1.f);
        EmitVertex();
        gl_Position =  ( pos+ vec4((xPosAdj+1)*barWidth,
                          0.0-glyphSize, 0.0, 0.0));
        position =vec2(1.f,0.f);
        EmitVertex();
    }

    //draw indicator bar
    pass = 1;
    barColor =  (color[0]);
    gl_Position = vec4( pos.x +halfVar*barWidth,
                        pos.y - glyphSize, pos.z,1.0);
```

```
    position =vec2(0.f,0.f);
    EmitVertex();
    EmitVertex();
    gl_Position = vec4( pos.x +halfVar*barWidth,
                        pos.y - glyphSize - barWidth,
                        pos.z,1.0);
    position =vec2(1.f,0.f);
    EmitVertex();

    gl_Position = vec4( pos.x - halfVar*barWidth,
                        pos.y-glyphSize, pos.z,1.0);
    position =vec2(0.f,1.f);
    EmitVertex();
    gl_Position = vec4( pos.x - halfVar*barWidth,
                        pos.y - glyphSize-barWidth,
                        pos.z,1.0);
    position =vec2(1.f,1.f);
    EmitVertex();
}
```

frag.glsl

```
#version 460

in float barColor;
in float pass;
in vec2 position;
uniform sampler1D attr_texture; //data map
uniform sampler1D cat_texture; //cat map
uniform mat4 matrix;
layout (location = 0) out vec4 color;

float ambient = 0.5;
float opacity = 1.0f;
float no_cat = 10.0f;
float lnCol = 0.5f;

void main() {
    vec3 Normal;
    Normal =vec3(position,0.f);
    vec4 colors;
    float barC =float(barColor/(no_cat + 1.0f));

    if (pass < 0.0) {
        color = vec4(lnCol, lnCol, lnCol, lnCol);
        return;
    } else if (pass < 1.0) {
        colors = texture( attr_texture, barColor);
        Normal.z= 1.f-2*(abs(position.x-0.5));
    } else {
        colors = texture( cat_texture, barC);
        Normal.xy = position * 2.0 - vec2(1.0);
        float mag = dot( Normal.xy, Normal.xy );
          Normal.z = 1.f-2*(abs(position.x-0.5));
        if (pass > 1.0) {
            if (mag > 1.0) {
                discard;
            }
            Normal.z = sqrt(1.0-mag);
        }
    }
    colors.w = opacity;

    vec4 temp = matrix * vec4(0.f,0.f,1.f,0.f);
    vec3 lightDir = normalize(temp.xyz);
    float diffuse = clamp(dot(lightDir, Normal),0,1);
    color = colors*(vec4(vec3(ambient),1.f) +
                    vec4(vec3(diffuse),1.f));
}
```