UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

Facultad de Ingeniería de Producción y Servicios

Ciencia de la Computación

Práctica 07

ALUMNOS:
Pfuturi Huisa, Oscar David
Quispe Menor, Hermogenes
Fernandez Mamani, Brayan Gino
Quiñonez Lopez, Efrain German
Santos Apaza, Yordy Williams

DOCENTE:
MSc. Vicente Machaca Arceda

CURSO:
Computación Gráfica

9 de junio de 2021

Índice

1. Implemente todos los ejemplos del capitulo II, del libro "Computer Graphics Programming in OpenGL with C++" [Gordon and Clevenger(2020)].

1.1. Programa 2.1: Creación de una ventana.

Nuestro main() incluye un ciclo de renderizado muy simple que llama a nuestra función display() repetidamente. Los parámetros del comando glfwCreateWindow() especifican el ancho y alto de la ventana (en píxeles) y el título colocado en la parte superior de la ventana. (Los dos parámetros adicionales que se establecen en NULL, y que no estamos usando, permiten el modo de pantalla completa y el uso compartido de recursos). La sincronización vertical (VSync) se habilita mediante los comandos glfwSwapInterval() y glfwSwapBuffers(), las ventanas GLFW tienen un búfer doble predeterminado.

```
#include <glad/glad.h>
#include <glfw/glfw3.h>
#include <iostream>

using namespace std;
void init(GLFWwindow* window) { }

void display(GLFWwindow* window, double currentTime) {
    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
int main(void) {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 -
   program1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)){ exit(
   EXIT_FAILURE); }
    glfwSwapInterval(1);
    init(window);
    while (!glfwWindowShouldClose(window)) {
            display(window, glfwGetTime());
            glfwSwapBuffers(window);
            glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

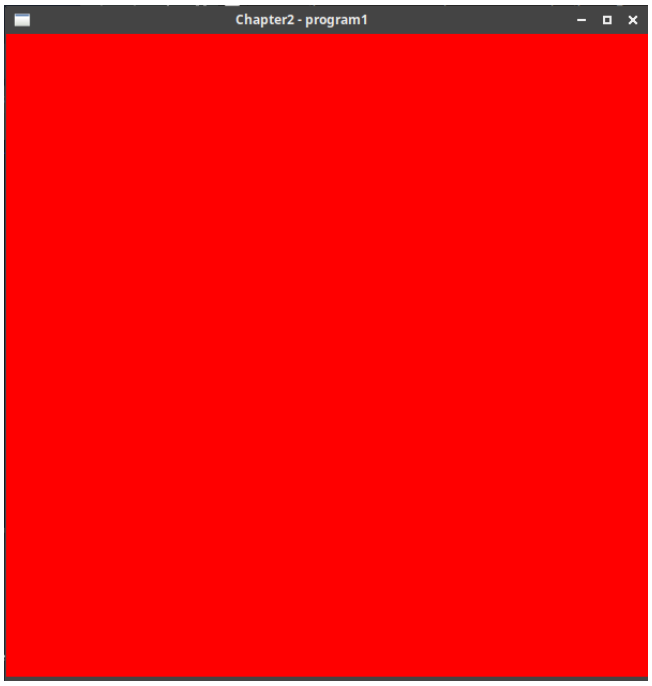

Figura 1: Ventana con fondo rojo

1.2. Programa 2.2: Shaders para la creación de punto (pixel).

la función CreateShaderProgram() implementado comienza declarando dos sombreadores como cadenas de caracteres llamadas vshaderSource y fshaderSource. Luego llama a lCreateShade() dos veces, lo que genera los dos sombreadores de tipos GL_VERTEX_ SHADER y GL_FRAGMENT_SHADER. Los shader proporcionan el código para ciertas etapas programables del pipeline de renderizado. También se pueden usar en una forma un poco más limitada para el cálculo general en la GPU.

```
#include <glad/glad.h>
#include <glfw/glfw3.h>
#include <iostream>

using namespace std;

#define numVAOs 1
GLuint renderingProgram;
GLuint vao[numVAOs];

GLuint createShaderProgram() {
    const char *vshaderSource =
            "#version 430 \n"
            "void main(void) \n"
            "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";
    // creamos un vertice, no especificamos salida, porque gl_position
    es por defecto de salida
    const char *fshaderSource =
            "#version 430 \n"
            "out vec4 color; \n"
```

```
            "void main(void) \n"
            "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";
    //en el etapss entre vertex y fragment, el vertice se convierte en un pixel
    //especificamos el color de los pixeles

    GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
    GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);

    glShaderSource(vShader, 1, &vshaderSource, NULL);
    glShaderSource(fShader, 1, &fshaderSource, NULL);
    glCompileShader(vShader);
    glCompileShader(fShader);
    GLuint vfProgram = glCreateProgram();
    glAttachShader(vfProgram, vShader);
    glAttachShader(vfProgram, fShader);
    glLinkProgram(vfProgram);
    return vfProgram;
}
void init(GLFWwindow* window) {
    renderingProgram = createShaderProgram();
    glGenVertexArrays(numVAOs, vao);
    glBindVertexArray(vao[0]);
}
void display(GLFWwindow* window, double currentTime) {
    glUseProgram(renderingProgram);
    glPointSize(30.0f); // un vertice es un pixel, con esto especificamos el tamanio del pixel
    glDrawArrays(GL_POINTS, 0, 1);

}
int main(void) {
    //El main es el mismo visto anteriormente
}
```

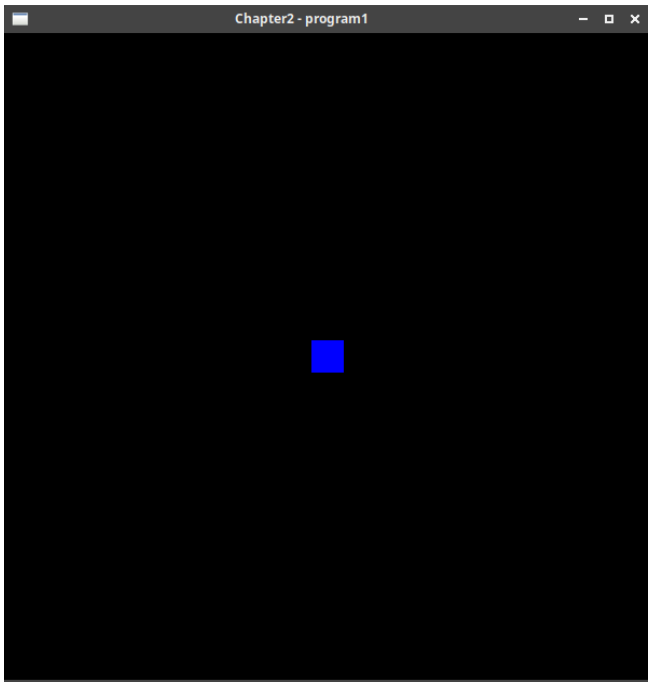

Figura 2: Ventana con GLFW

1.3. Programa 2.3: Obtención de GLSL errores.

Se implemneto las siguientes funciones para obtener errores de GLSL:

- checkOpenGLError: comprueba el indicador de error de OpenGL para detectar la aparición de un error de OpenGL.
- printShaderLog: muestra el contenido del registro de OpenGL cuando falla la compilación de GLSL.
- printProgramLog: muestra el contenido del registro de OpenGL cuando falla el enlace GLSL.

La funcion checkOpenGLError(), es útil para detectar errores de compilación GLSL y errores de tiempo de ejecución de OpenGL, por lo que se recomienda encarecidamente utilizarlo en una aplicación C++ / OpenGL durante el desarrollo.

```
1  #include <glad/glad.h>
2  #include <glfw/glfw3.h>
3  #include <iostream>
4  using namespace std;
5
6  #define numVAOs 1
7  GLuint renderingProgram;
8  GLuint vao[numVAOs];
9
10 // functions to catch errors in GLSL
11
12 void printShaderLog(GLuint shader) {
13     int len = 0;
```

```
14      int chWrittn = 0;
15      char *log;
16      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
17      if (len > 0) {
18              log = (char *)malloc(len);
19              glGetShaderInfoLog(shader, len, &chWrittn, log);
20              cout << "Shader Info Log: " << log << endl;
21              free(log);
22      }
23  }
24  void printProgramLog(int prog) {
25      int len = 0;
26      int chWrittn = 0;
27      char *log;
28      glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
29      if (len > 0) {
30              log = (char *)malloc(len);
31              glGetProgramInfoLog(prog, len, &chWrittn, log);
32              cout << "Program Info Log: " << log << endl;
33              free(log);
34      }
35  }
36  bool checkOpenGLError() {
37      bool foundError = false;
38      int glErr = glGetError();
39      while (glErr != GL_NO_ERROR) {
40              cout << "glError: " << glErr << endl;
41              foundError = true;
42              glErr = glGetError();
43      }
44      return foundError;
45  }
46  ////////////////////
47  ////////////////////
48
49  GLuint createShaderProgram() {
50      GLint vertCompiled;
51      GLint fragCompiled;
52      GLint linked;
53
54      const char *vshaderSource =
55              "#version 430 \n"
56              "void main(void) \n"
57              "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";
58      // creamos un vertice, no especificamos salida, porque gl_position
        es por defecto de salida
59      const char *fshaderSource =
60              "#version 430 \n"
61              "out vec4 color; \n"
```

```
62              "void main(void) \n"
63              "{ if (gl_FragCoord.x < 200) color = vec4(1.0, 0.0, 0.0,
    1.0); else color = vec4(0.0, 0.0, 1.0, 1.0); }";
64      //especificamos el color de los pixeles
65
66      GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
67      GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
68
69      glShaderSource(vShader, 1, &vshaderSource, NULL);
70      glShaderSource(fShader, 1, &fshaderSource, NULL);
71
72      glCompileShader(vShader);
73      checkOpenGLError();
74      glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
75      if (vertCompiled != 1) {
76              cout << "vertex compilation failed" << endl;
77              printShaderLog(vShader);
78      }
79
80      glCompileShader(fShader);
81      checkOpenGLError();
82      glGetShaderiv(fShader, GL_COMPILE_STATUS, &fragCompiled);
83      if (fragCompiled != 1) {
84              cout << "fragment compilation failed" << endl;
85              printShaderLog(fShader);
86      }
87
88
89      GLuint vfProgram = glCreateProgram();
90      glAttachShader(vfProgram, vShader);
91      glAttachShader(vfProgram, fShader);
92
93      glLinkProgram(vfProgram);
94      checkOpenGLError();
95      glGetProgramiv(vfProgram, GL_LINK_STATUS, &linked);
96      if (linked != 1) {
97              cout << "linking failed" << endl;
98              printProgramLog(vfProgram);
99      }
100
101     return vfProgram;
102 }
103
104 void init(GLFWwindow* window) {
105  //igual al implementado anteriormente
106 }
107
108 void display(GLFWwindow* window, double currentTime) {
109     glUseProgram(renderingProgram);
```

```
110     glPointSize(400.0f); // un vertice es un pixel, con esto
        especificamos el tamanio del pixel
111     //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); //wire frame
112     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // noraml
113     glDrawArrays(GL_POINTS, 0, 1);
114
115 }
116
117 int main(void) {
118     //igual al implementado anteriormente
119 }
```

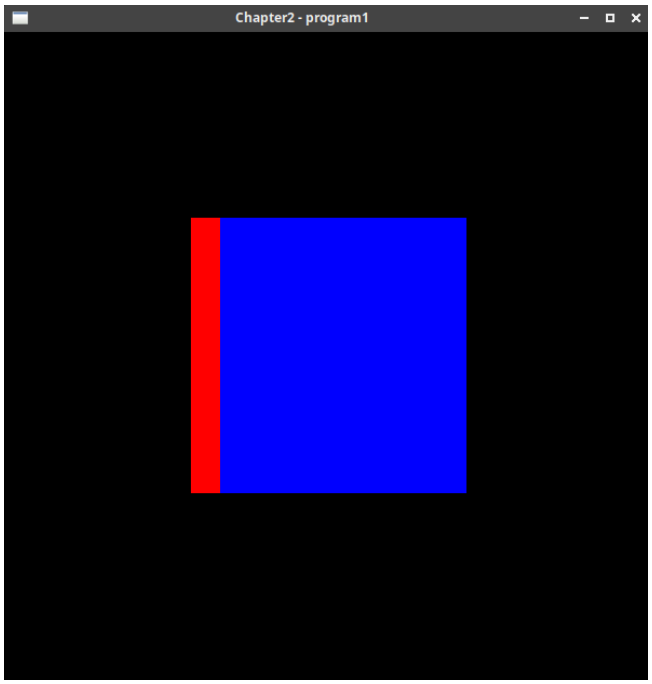

Figura 3: Ventana con

1.4. Programa 2.4: Lectura de los shaders desde un archivo.

El código para leer sombreadores se proporciona en readShaderSource(). Lee el archivo de texto del sombreador y devuelve una matriz de cadenas, donde cada cadena es una línea de texto del archivo. Luego determina el tamaño de esa matriz en función de la cantidad de líneas leídas. En este ejemplo, el vertes y fragment shader ahora se coloca en los archivos de texto “vertShader.glsl” y “fragShader.glsl” respectivamente.

```
1 #include <glad/glad.h>
2 #include <glfw/glfw3.h>
3
4 #include <iostream>
5 #include <fstream>
6 #include <unistd.h>
7 #include <cstring>
```

```
8  using namespace std;
9
10 char* vertShaderPath = "/home/hermogene/Documents/ComputacionGrafica/
       opengl/src/04_shader_file/vertShader.glsl";
11 char* fragShaderPath = "/home/hermogene/Documents/ComputacionGrafica/
       opengl/src/04_shader_file/fragShader.glsl";
12 #define numVAOs 1
13 GLuint renderingProgram;
14 GLuint vao[numVAOs];
15
16 string readShaderSource(const char *filePath) {
17     string content;
18     ifstream fileStream(filePath, ios::in);
19     string line = "";
20     while (!fileStream.eof()) {
21             getline(fileStream, line);
22             content.append(line + "\n");
23     }
24     fileStream.close();
25     return content;
26 }
27
28 ///////////////////////
29 // functions to catch errors in GLSL
30 ///////////////////////
31 void printShaderLog(GLuint shader) {
32     int len = 0;
33     int chWrittn = 0;
34     char *log;
35     glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
36     if (len > 0) {
37             log = (char *)malloc(len);
38             glGetShaderInfoLog(shader, len, &chWrittn, log);
39             cout << "Shader Info Log: " << log << endl;
40             free(log);
41     }
42 }
43 void printProgramLog(int prog) {
44     int len = 0;
45     int chWrittn = 0;
46     char *log;
47     glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
48     if (len > 0) {
49             log = (char *)malloc(len);
50             glGetProgramInfoLog(prog, len, &chWrittn, log);
51             cout << "Program Info Log: " << log << endl;
52             free(log);
53     }
54 }
```

```
55 bool checkOpenGLError() {
56     bool foundError = false;
57     int glErr = glGetError();
58     while (glErr != GL_NO_ERROR) {
59             cout << "glError: " << glErr << endl;
60             foundError = true;
61             glErr = glGetError();
62     }
63     return foundError;
64 }
65 ////////////////////////
66 ////////////////////////
67
68 GLuint createShaderProgram() {
69     GLint vertCompiled;
70     GLint fragCompiled;
71     GLint linked;
72
73     GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
74     GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
75
76     // read shaders from files
77     string vertShaderStr = readShaderSource(vertShaderPath);
78     string fragShaderStr = readShaderSource(fragShaderPath);
79     const char *vertShaderSrc = vertShaderStr.c_str();
80     const char *fragShaderSrc = fragShaderStr.c_str();
81     glShaderSource(vShader, 1, &vertShaderSrc, NULL);
82     glShaderSource(fShader, 1, &fragShaderSrc, NULL);
83
84     glCompileShader(vShader);
85     checkOpenGLError();
86     glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
87     if (vertCompiled != 1) {
88             cout << "vertex compilation failed" << endl;
89             printShaderLog(vShader);
90     }
91
92     glCompileShader(fShader);
93     checkOpenGLError();
94     glGetShaderiv(fShader, GL_COMPILE_STATUS, &fragCompiled);
95     if (fragCompiled != 1) {
96             cout << "fragment compilation failed" << endl;
97             printShaderLog(fShader);
98     }
99
100
101     GLuint vfProgram = glCreateProgram();
102     glAttachShader(vfProgram, vShader);
103     glAttachShader(vfProgram, fShader);
```

```
104
105     glLinkProgram(vfProgram);
106     checkOpenGLError();
107     glGetProgramiv(vfProgram, GL_LINK_STATUS, &linked);
108     if (linked != 1) {
109             cout << "linking failed" << endl;
110             printProgramLog(vfProgram);
111     }
112
113     return vfProgram;
114 }
115
116 void init(GLFWwindow* window) {
117     renderingProgram = createShaderProgram();
118     glGenVertexArrays(numVAOs, vao);
119     glBindVertexArray(vao[0]);
120 }
121
122 void display(GLFWwindow* window, double currentTime) {
123     glUseProgram(renderingProgram);
124     glPointSize(400.0f); // un vertice es un pixel, con esto
    especificamos el tamanio del pixel
125     //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); //wire frame
126     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // noraml
127     glDrawArrays(GL_POINTS, 0, 1);
128
129 }
130
131 int main(void) {
132     if (!glfwInit()) { exit(EXIT_FAILURE); }
133     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
134     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
135
136     GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 -
    program1", NULL, NULL);
137     glfwMakeContextCurrent(window);
138
139     //if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
140     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)){ exit(
    EXIT_FAILURE); }
141
142     glfwSwapInterval(1);
143     init(window);
144     while (!glfwWindowShouldClose(window)) {
145             display(window, glfwGetTime());
146             glfwSwapBuffers(window);
147             glfwPollEvents();
148     }
149
```

```
150     glfwDestroyWindow(window);
151     glfwTerminate();
152     exit(EXIT_SUCCESS);
153
154 }
```

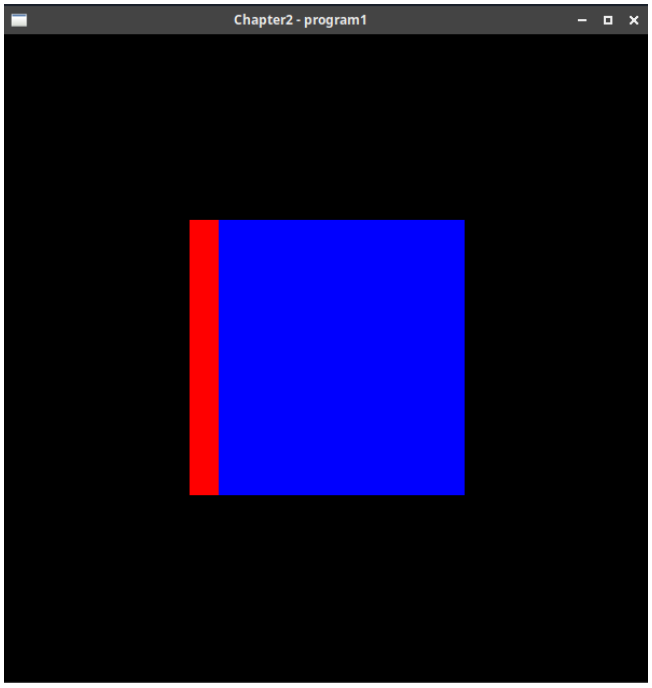

Figura 4: Caption

1.5. Programa 2.5: Un triangulo

En OpenGL, específicamente en la llamada glDrawArrays(), especificamos GL_TRIANGLES (en lugar de GL_POINTS), y también especificamos que hay tres vértices enviados a través de la canalización. Esto hace que el sombreador de vértices se ejecute tres veces y, en cada iteración, la variable incorporada gl_VertexID se incrementa automáticamente (inicialmente se establece en 0). Al probar el valor de gl_VertexID, el sombreador está diseñado para generar un punto diferente cada una de las tres veces que se ejecuta. Recuerde que los tres puntos luego pasan por la etapa de rasterización, produciendo un triángulo relleno.

```
1  #include <glad/glad.h>
2  #include <glfw/glfw3.h>
3  #include <iostream>
4
5  void framebuffer_size_callback(GLFWwindow* window, int width, int
       height);
6  void processInput(GLFWwindow *window);
7
8  // settings
9  const unsigned int SCR_WIDTH = 800;
10 const unsigned int SCR_HEIGHT = 600;
```

```
11
12  const char *vertexShaderSource = "#version 330 core\n"
13      "layout (location = 0) in vec3 aPos;\n"
14      "out vec4 vertexColor;\n"
15      "void main()\n"
16      "{\n"
17      "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
18      "   vertexColor = vec4(0.0, 0.0, 1.0, 1.0); \n"
19      "}\0";
20
21  // fragment shader, esta escrito en GLSL
22  const char *fragmentShaderSource = "#version 330 core\n"
23      "out vec4 FragColor;\n"
24      "in vec4 vertexColor;\n"
25      "void main()\n"
26      "{\n"
27      "   //FragColor = vec4(0.0, 0.0, 1.0, 1.0);\n"
28      "   FragColor = vertexColor;\n"
29      "}\n\0";
30
31  int main(){
32
33      float offset;
34      // glfw: inicializar y configurar
35      glfwInit();
36      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
37      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
38      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
39
40      #ifdef __APPLE__
41      glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
42      #endif
43      // glfw creacion de ventana
44      // --------------------
45      GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "
    LearnOpenGL", NULL, NULL);
46      if (window == NULL)
47      {
48          std::cout << "Failed to create GLFW window" << std::endl;
49          glfwTerminate();
50          return -1;
51      }
52
53      glfwMakeContextCurrent(window);
54      glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
55      // glad: load all OpenGL function pointers
56
57      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)){
58          std::cout << "Failed to initialize GLAD" << std::endl;
```

```
59          return -1;
60      }
61
62      // vertex shader
    ////////////////////////////////////////////////
63      int vertexShader = glCreateShader(GL_VERTEX_SHADER);
64      glShaderSource(vertexShader, 1, &vertexShaderSource, NULL); //
    asignamos el codigo del shader
65      glCompileShader(vertexShader);
66
67      // check for shader compile errors
68      int success;
69      char infoLog[512];
70      glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
71      if (!success){
72          glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
73          std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
    infoLog << std::endl;
74      }
75
76      // fragment shader ////////////////////////////////////////////
77      int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
78      glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
79      glCompileShader(fragmentShader);
80      // check for shader compile errors
81      glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
82      if (!success){
83          glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
84          std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n"
    << infoLog << std::endl;
85      }
86
87      // link shaders
88      int shaderProgram = glCreateProgram();
89      glAttachShader(shaderProgram, vertexShader);
90      glAttachShader(shaderProgram, fragmentShader);
91      glLinkProgram(shaderProgram);
92      // check for linking errors
93      glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
94      if (!success) {
95          glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
96          std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" <<
    infoLog << std::endl;
97
98      }
99      glDeleteShader(vertexShader);
100     glDeleteShader(fragmentShader);
101
102     // triangle
```

```
    ------------------------------------------------------------------
103  float vertices[] = {
104
105      -0.25f, -0.5f, 0.0f,
106      0.25f, -0.5f, 0.0f,
107      0.0f,  0.8f, 0.0f,
108
109  };
110
111  unsigned int VBO, VAO;
112  glGenVertexArrays(1, &VAO);
113  glGenBuffers(1, &VBO); //separamos memoria en el GPU para los
     vertices
114  glBindVertexArray(VAO);
115
116  glBindBuffer(GL_ARRAY_BUFFER, VBO); // enlazamos la memoria con
     los vertices. GL_ARRAY_BUFFER refiere a el tipo de dato vertice
117  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
     GL_STATIC_DRAW); // copiamos los datos
118
119  //GL_STREAM_DRAW: los datos se configuran solo una vez y la GPU
     los utiliza como maximo unas pocas veces.
120  //GL_STATIC_DRAW: los datos se establecen solo una vez y se
     utilizan muchas veces.
121  //GL_DYNAMIC_DRAW: los datos se cambian mucho y se utilizan muchas
      veces.
122
123  // /triangle
     ------------------------------------------------------------------
124  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
     (void*)0);
125  glEnableVertexAttribArray(0);
126
127  // tenga en cuenta que esto esta permitido, la llamada a
     glVertexAttribPointer registro VBO como el objeto de bufer de
     vertice enlazado del atributo de vertice, por lo que luego podemos
     desvincular de forma segura.
128  glBindBuffer(GL_ARRAY_BUFFER, 0);
129  glBindVertexArray(0);
130
131  while (!glfwWindowShouldClose(window))
132  {
133      // input
134      processInput(window);
135
136      // render
137      glClearColor(0.0, 0.0, 0.0, 1.0);
138      glClear(GL_COLOR_BUFFER_BIT);
139
```

```
140         // dibuja nuestro triangulo
141         glUseProgram(shaderProgram);
142         glBindVertexArray(VAO); // Dado que solo tenemos un VAO, no es
     necesario vincularlo cada vez, pero lo haremos para mantener las
    cosas un poco mas organizadas.
143         glDrawArrays(GL_TRIANGLES, 0, 3); // 0:vertice inicio, 3:
    cuantos vertices dibujar
144         glfwSwapBuffers(window);
145         glfwPollEvents();
146     }
147
148     glDeleteVertexArrays(1, &VAO);
149     glDeleteBuffers(1, &VBO);
150
151     //glDeleteBuffers(1, &EBO);
152     glDeleteProgram(shaderProgram);
153
154     // glfw: terminate, borrando todos los recursos GLFW previamente
    asignados.
155     glfwTerminate();
156     return 0;
157 }
158
159 // process all input: consultar GLFW si las teclas relevantes se
    presionaron / liberaron este marco y reaccionar en consecuencia
160 void processInput(GLFWwindow *window)
161
162 {
163     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
164         glfwSetWindowShouldClose(window, true);
165 }
166
167 // glfw: cada vez que cambia el tamanio de la ventana (por el sistema
    operativo o el tamanio del usuario), esta funcion de devolucion de
    llamada se ejecuta
168 void framebuffer_size_callback(GLFWwindow* window, int width, int
    height)
169
170 {
171     glViewport(0, 0, width, height);
172 }
```

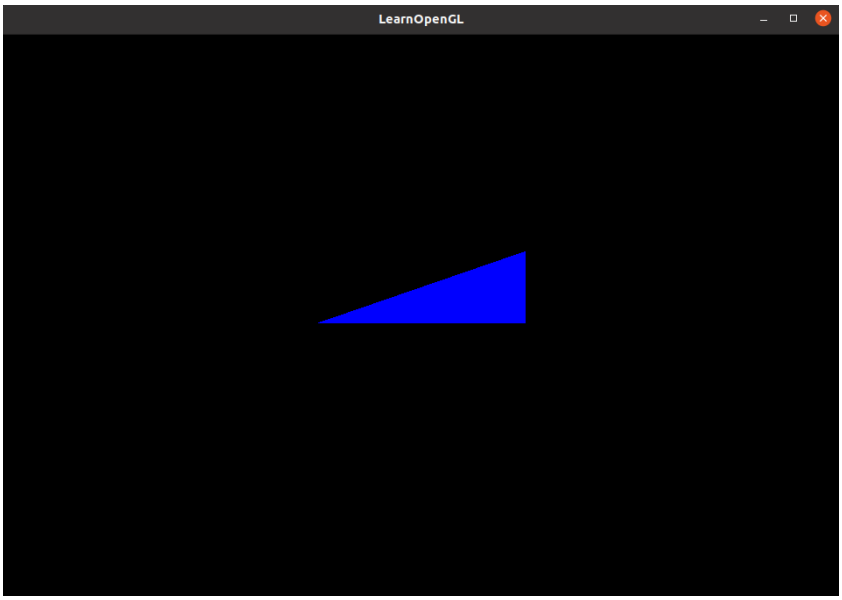

Figura 5: Triangulo

1.6. Programa 2.6: Una animación simple

Anteriormente se construyo el main() para hacer una sola llamada a init(), y luego para llamar a display() repetidamente. Por lo tanto, si bien cada uno de los ejemplos anteriores puede parecer una única escena renderizada fija, en realidad el bucle principal estaba provocando que se dibujara una y otra vez. Por esta razón, nuestro main() ya está estructurado para admitir animación. Simplemente diseñamos nuestra función display() para alterar lo que dibuja con el tiempo. Cada representación de nuestra escena se denomina fotograma y la frecuencia de las llamadas a display() es la velocidad de fotogramas. El manejo de la tasa de movimiento dentro de la lógica de la aplicación se puede controlar usando el tiempo transcurrido desde el cuadro anterior (esta es la razón para incluir "currentTime" como parámetro en la función display()).

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

void display();
void reshape(int, int);
void timer(int);
int state = 1;

void init()
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
}

int main(int argc, char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

```
19
20      glutInitWindowPosition(200,100);
21      glutInitWindowSize(800,600);
22
23      glutCreateWindow("ventana");
24
25      glutDisplayFunc(display);
26      glutReshapeFunc(reshape);
27      glutTimerFunc(0,timer,0);
28      init();
29
30      glutMainLoop();
31  }
32
33  float x_position = 0.0;
34
35  void display()
36  {
37      glClear(GL_COLOR_BUFFER_BIT);
38      glLoadIdentity();
39
40      glBegin(GL_POLYGON);
41
42      glVertex2f(x_position, -1.0);
43      glVertex2f(x_position + 3.0, -1.0);
44      glVertex2f(x_position + 3.0, 1.0);
45
46      glEnd();
47      //glFlush();
48      glutSwapBuffers();
49  }
50
51  void reshape(int w, int h)
52  {
53      glViewport(0,0,(GLsizei)w, (GLsizei)h);
54      glMatrixMode(GL_PROJECTION);
55      glLoadIdentity();
56      gluOrtho2D(-10,10,-10,10);
57      glMatrixMode(GL_MODELVIEW);
58  }
59
60  void timer(int)
61  {
62      glutPostRedisplay();
63      glutTimerFunc(1000/60,timer,0);
64
65      if(state == 1){
66          if(x_position < 7)
67              x_position += 0.15;
```

```
68          else
69              state = -1;
70      }
71      else{
72          if(x_position> -10)
73              x_position-= 0.15;
74          else
75              state = 1;
76      }
77  }
```

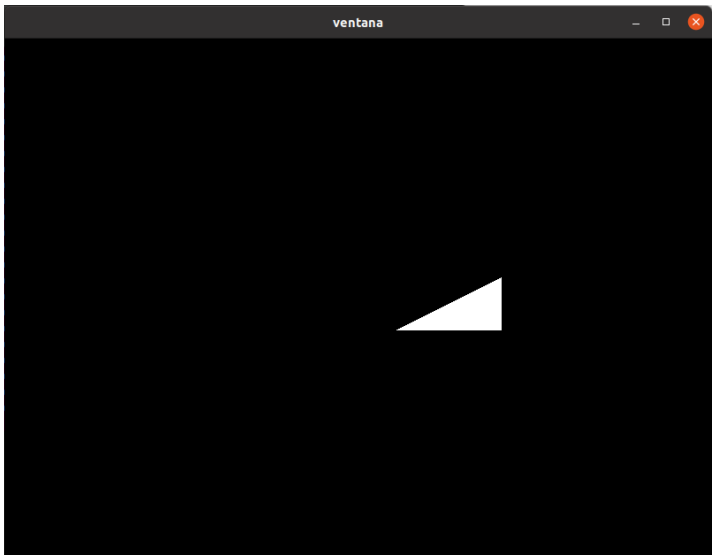

Figura 6: animación simple

Figura 7: animación simple

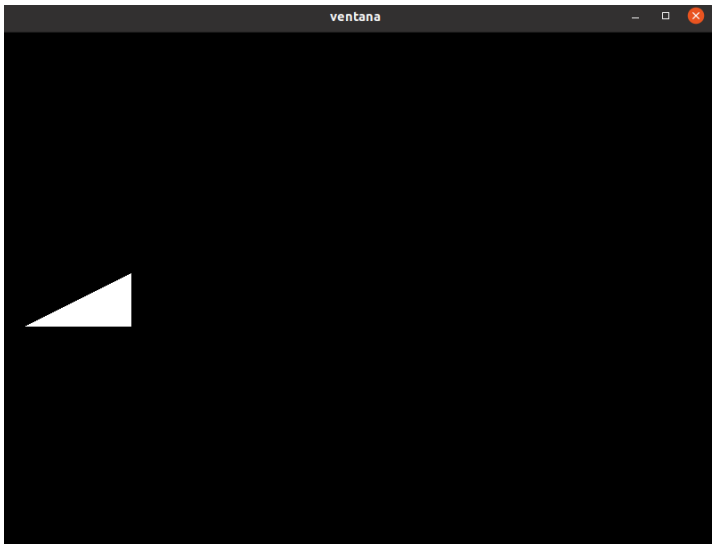

Figura 8: animación simple

2. Modifique el programa 2.2, para agregar una animación donde el punto crezca y se encoja (puede utilizar glPointSize()).

Para la realización de este problema vamos a reutilizar y modificar el problema 2.6, lo siguiente es realizar la parte creciente, para esto debemos tener en cuento que en el cuadrante 1 se incrementa en el eje X y Y, para el segundo cuadrante se disminuye en Y y se incremente en X, para el tercer cuadrante se disminuye en X y Y, por ultimo para el cuarto cuadrante se aumenta en X y se disminuye en Y. Para la parte decreciente simplemente se cambia de signo a nuestra variable $x - position$.

```
1
2  void display()
3  {
4      glClear(GL_COLOR_BUFFER_BIT);
5      glLoadIdentity();
6
7      glBegin(GL_POLYGON);
8
9      glVertex2f(-1.0-x_position, x_position+1.0);
10     glVertex2f(x_position+1.0, x_position+1.0);
11     glVertex2f(x_position+1.0, -1.0-x_position);
12     glVertex2f(-1.0-x_position,-1.0-x_position);
13
14     glEnd();
15     glutSwapBuffers();
16 }
```

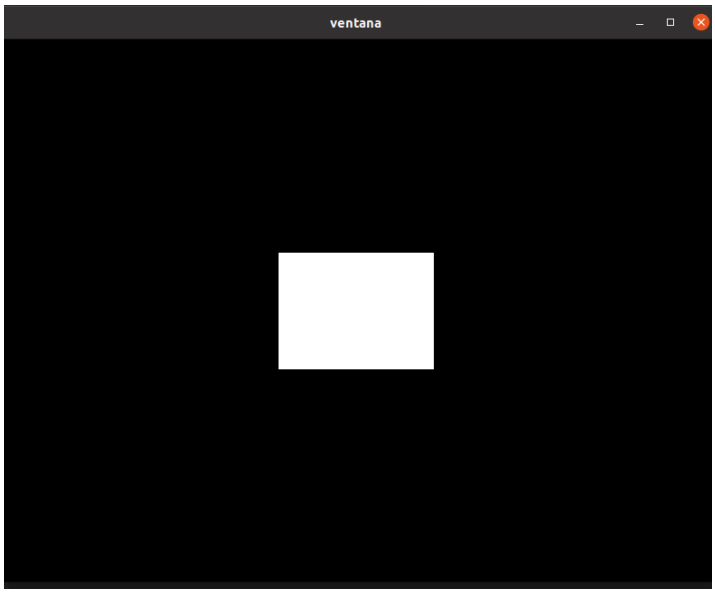

Figura 9: Píxel creciente y decreciente

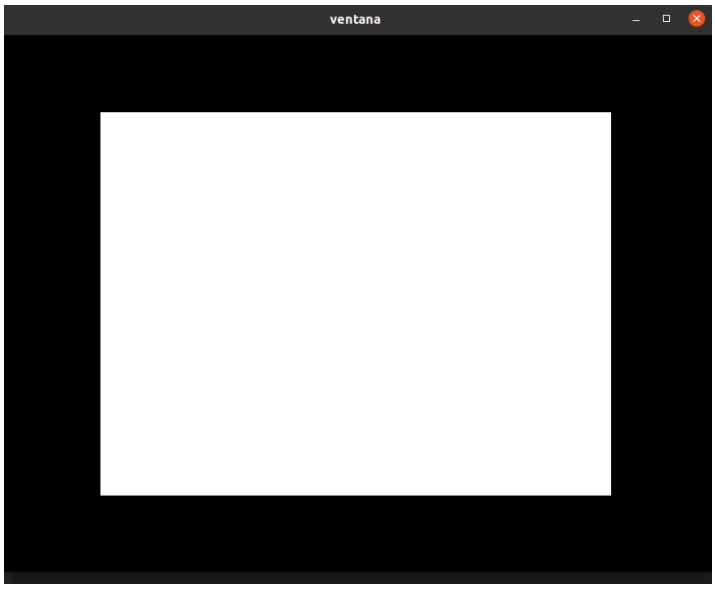

Figura 10: Píxel creciente y decreciente

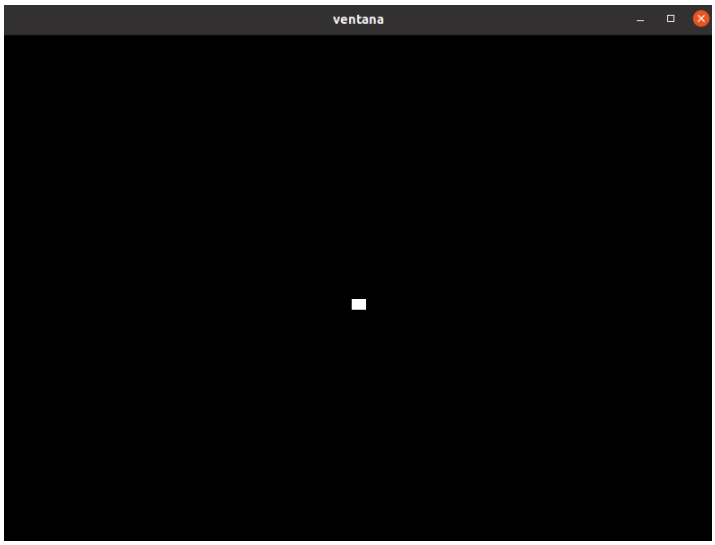

Figura 11: Píxel creciente y decreciente

3. Modifique el programa 2.5, de manera tal que se grafique un triangulo isósceles.

Simplemente se modifica los vértices del ejercicio 1.5 para obtener un triangulo isósceles.

```
1
2     // triangulo isosceles--
3     float vertices[] = {
4
5         -0.25f, -0.5f, 0.0f,
6         0.25f, -0.5f, 0.0f,
7         0.0f,  0.8f, 0.0f,
8
9     };
```

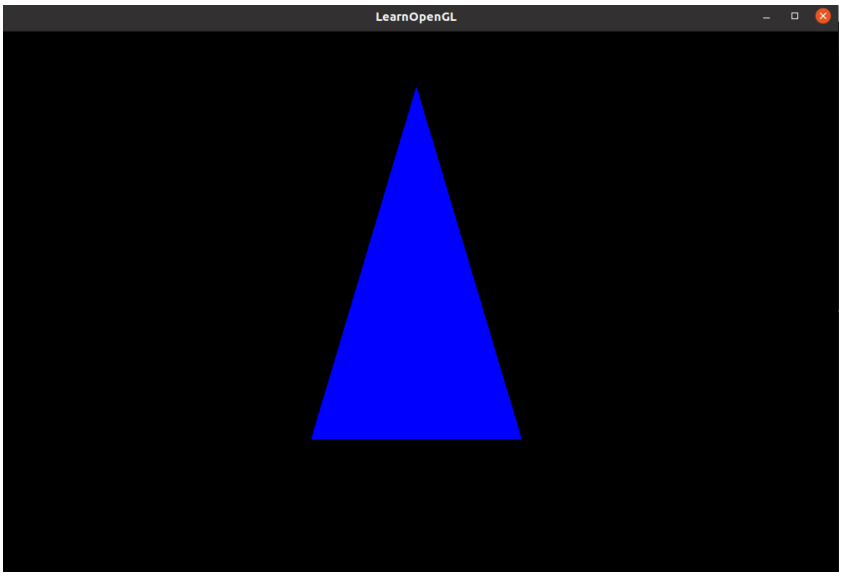

Figura 12: Triangulo Isósceles

4. Enlace

El código fuente está disponible en: https://github.com/oscar-pfuturi-h/Comp-Grafica/tree/main/practica_07

Referencias

[1] Lab. de Visualización y Computación Gráfica. (s. f.). Escenas 3D. Computación Grafica. Recuperado 15 de mayo de 2021, de http://www.cs.uns.edu.ar/cg/clasespdf/3-Pipe3D.pdf