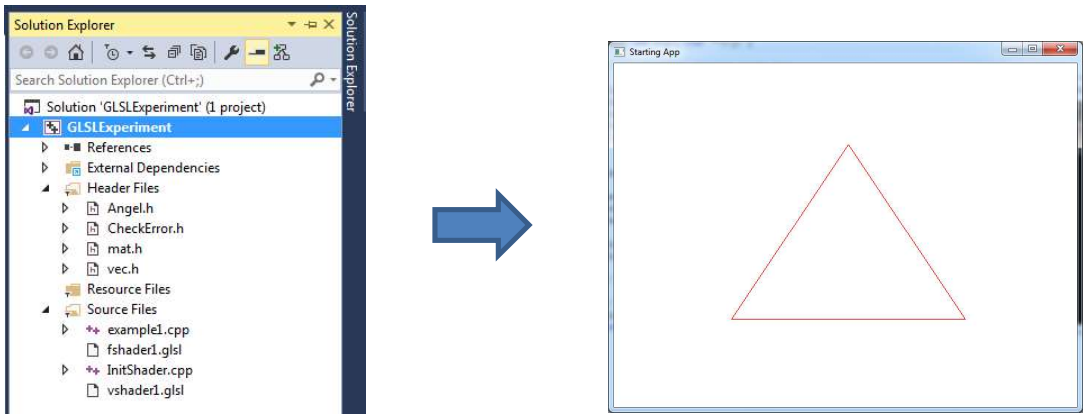
BÀI THỰC HÀNH SỐ 1 – LẬP TRÌNH OPENGL MÔ HÌNH HIỆN ĐẠI

A – Trải nghiệm một ví dụ

- Đã cài đặt Microsoft Visual Studio - C++, khuyến cáo phiên bản 2017, có thể dùng bản Community – bản miễn phí.
- Download project được gửi kèm hướng dẫn và trải nén.
- Project **đã tích hợp cài OpenGL, Glew, Glut** (*Để hiểu về thiết lập từ đầu → đọc tài liệu hướng dẫn* "Bài thực hành số 1 – Hướng dẫn cài đặt OpenGL, Glew, Glut").
- Chạy thử chương trình, kết quả như Hình 1.1.

Cấu trúc Project và kết quả



Hình 1.1

B – Code của project

Yêu cầu: Đọc lướt – chưa cần hiểu ➔ đọc mục C

1. Nội dung file example1.cpp

```
//Chương trình vẽ 1 tam giác theo mô hình lập trình OpenGL hiện đại

#include "Angel.h"  /* Angel.h là file tự phát triển (tác giả Prof. Angel), có
chứa cả khai báo includes glew và freeglut*/

using namespace std;

// Cấu trúc các hàm trong file
void generateGeometry( void );
void initGPUBuffers( void );
void shaderSetup( void );
void display( void );
void keyboard( unsigned char key, int x, int y );

GLuint program; //Biến quản lý đối tượng program

typedef vec2 point2;
```

```
// Số các đỉnh của tập các đoạn thẳng
const int NumPoints = 3;
// Mảng các đỉnh của hình cần vẽ
point2 points[NumPoints];

void generateGeometry( void )
{
      // Gán giá trị các đỉnh cho mảng
      points[0] = point2( -0.5, -0.5 );
      points[1] = point2( 0.0, 0.5 );
      points[2] = point2( 0.5, -0.5 );
}

void initGPUBuffers( void )
{
    // Tạo một VAO - vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Tạo và khởi tạo một buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );
}

void shaderSetup( void )
{
      // Nạp các shader và sử dụng chương trình shader
    program = InitShader( "vshader1.glsl", "fshader1.glsl" );
                                /* hàm InitShader khai báo trong Angel.h*/
    glUseProgram( program );

    // Khởi tạo thuộc tính vị trí đỉnh từ vertex shader
    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    glClearColor( 1.0, 1.0, 1.0, 1.0 ); /* Thiết lập màu trắng là màu xóa màn
hình*/
}
```

```
void display( void )
{
    /* Xóa bộ đệm màu màn hình bằng màu thiết lập bởi glClearColor() - Xóa màn
hình*/
    glClear( GL_COLOR_BUFFER_BIT );

    glDrawArrays( GL_LINE_LOOP, 0, NumPoints );    /*Vẽ các đoạn thẳng*/
    glFlush();     /* Đẩy việc thực thi các lệnh OpenGL đến phần cứng đồ họa - thực
thi lệnh OpenGL trong thời gian hữu hạn*/
}

void keyboard( unsigned char key, int x, int y )
{
    // keyboard handler
    switch ( key ) {
    case 033:            // 033 is Escape key octal value
        exit(1);         // quit program
        break;
    }
}

int main( int argc, char **argv )
{
   // main function: program starts here
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB );
    glutInitWindowSize( 640, 480 );
    glutInitWindowPosition(100,150);
    glutCreateWindow( "Starting App" );

    glewInit();          /* Khởi tạo glew */

    generateGeometry( );  /* Gọi hàm sinh các điểm để vẽ */
    initGPUBuffers( );    /* Tạo GPU buffers*/
    shaderSetup( );       /* Kết nối file .cpp hiện tại với các file shader */

    glutDisplayFunc( display );    // Đăng ký hàm gọi lại về hiển thị
    glutKeyboardFunc( keyboard );  // Đăng ký hàm gọi lại cho sự kiện bàn phím

      /* Có thể thêm các hàm gọi lại về điều khiển chuột, thay đổi kích cỡ cửa sổ,
... ở đây*/

    glutMainLoop();   // Đưa vào vòng lặp vô hạn
    return 0;
}
```

2. Nội dung file InitShader.cpp

```
#include "Angel.h"

void printShaderInfoLog(GLuint obj); //Bổ sung - hàm in thông tin debug shader
void printProgramInfoLog(GLuint obj);//Bổ sung - hàm in thông tin debug program
static char* readShaderSource(const char* shaderFile);
GLuint InitShader(const char* vShaderFile, const char* fShaderFile);

namespace Angel {

/* Đọc file shaderFile lưu vào một biến chuỗi và thêm kí tự NULL ('\0') vào cuối
chuỗi, trả địa chỉ chuỗi ra tên hàm */
static char*
readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");

    if ( fp == NULL ) { return NULL; }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '\0';
    fclose(fp);

    return buf;
}


// Tạo một đối tượng chương trình GLSL gồm vertex và fragment shader
GLuint
InitShader(const char* vShaderFile, const char* fShaderFile)
{
    struct Shader {
      const char*  filename;
      GLenum       type;
      GLchar*      source;  // Lưu địa chỉ mã nguồn của mỗi file shader
    }  shaders[2] = {
      { vShaderFile, GL_VERTEX_SHADER, NULL },
      { fShaderFile, GL_FRAGMENT_SHADER, NULL }
```

```
    };

    GLuint program = glCreateProgram();

    for ( int i = 0; i < 2; ++i ) {

      Shader& s = shaders[i];

      /* Lưu địa chỉ của biến chuỗi chứa mã nguồn của file shader tương ứng vào
                                          mảng shaders */
      s.source = readShaderSource( s.filename ); /*Đọc nội dung file shader*/

      if ( shaders[i].source == NULL ) {
          std::cerr << "Failed to read " << s.filename << std::endl;
          exit( EXIT_FAILURE );
      }

      GLuint shader = glCreateShader( s.type );
      glShaderSource( shader, 1, (const GLchar**) &s.source, NULL );

//Biên dịch shaders và kiểm tra lỗi
      glCompileShader( shader );

      printShaderInfoLog(shader);
      GLint  compiled;
      glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
      if ( !compiled ) {
          std::cerr << s.filename << " failed to compile:" << std::endl;
          GLint  logSize;
          glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &logSize );
          char* logMsg = new char[logSize];
          glGetShaderInfoLog( shader, logSize, NULL, logMsg );
          std::cerr << logMsg << std::endl;
          delete [] logMsg;

          exit( EXIT_FAILURE );
      }

      delete [] s.source; // Giải phóng biến s.source

      glAttachShader( program, shader );
    }

    /* link  and error check */
    glLinkProgram(program);
```

```
    GLint  linked;
    glGetProgramiv( program, GL_LINK_STATUS, &linked );
    if ( !linked ) {
      std::cerr << "Shader program failed to link" << std::endl;
      GLint  logSize;
      glGetProgramiv( program, GL_INFO_LOG_LENGTH, &logSize);
      char* logMsg = new char[logSize];
      glGetProgramInfoLog( program, logSize, NULL, logMsg );
      std::cerr << logMsg << std::endl;
      delete [] logMsg;

      exit( EXIT_FAILURE );
    }

    /* use program object */
    glUseProgram(program);

    printProgramInfoLog(program);

    return program;
}
}  // Close namespace Angel block

// Cài đặt các hàm bổ sung

//Got this from http://www.lighthouse3d.com/opengl/glsl/index.php?oglinfo
// it prints out shader info (debugging!)
void printShaderInfoLog(GLuint obj)
{
      int infologLength = 0;
      int charsWritten = 0;
      char *infoLog;
      glGetShaderiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
      if (infologLength > 0)
      {
            infoLog = (char *)malloc(infologLength);
            glGetShaderInfoLog(obj, infologLength, &charsWritten, infoLog);
            printf("printShaderInfoLog: %s\n", infoLog);
            free(infoLog);
      }
      else {
            printf("Shader Info Log: OK\n");
      }
}
```

```
//Got this from http://www.lighthouse3d.com/opengl/glsl/index.php?oglinfo
// it prints out shader info (debugging!)
void printProgramInfoLog(GLuint obj)
{
	int infologLength = 0;
	int charsWritten = 0;
	char *infoLog;
	glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
	if (infologLength > 0)
	{
		infoLog = (char *)malloc(infologLength);
		glGetProgramInfoLog(obj, infologLength, &charsWritten, infoLog);
		printf("printProgramInfoLog: %s\n", infoLog);
		free(infoLog);
	}
	else {
		printf("Program Info Log: OK\n");
	}
}
```

3. Nội dung file vshader1.glsl

```
#version 400
in vec4 vPosition;
void
main()
{
    gl_Position = vPosition;
}
```

4. Nội dung file fshader1.glsl

```
#version 400

out vec4  fColor;

void
main()
{
    fColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

5. Nội dung file Angel.h

```
//////////////////////////////////////////////////////////////////////////////
//
```

```
//  --- Angel.h ---
//
//   The main header file for all examples from Angel 6th Edition
//
////////////////////////////////////////////////////////////////////////////

#ifndef __ANGEL_H__
#define __ANGEL_H__

// Visual Studio warns about deprecated fopen. Get rid of that warning
#define _CRT_SECURE_NO_WARNINGS

//----------------------------------------------------------------------------
//
// --- Include system headers ---
//

#include <cmath>
#include <iostream>

//  Define M_PI in the case it's not defined in the math header file
#ifndef M_PI
#  define M_PI  3.14159265358979323846
#endif

//----------------------------------------------------------------------------
//
// --- Include OpenGL header files and helpers ---
//
//   The location of these files vary by operating system.  We've included
//     copies of open-soruce project headers in the "GL" directory local
//     this this "include" directory.
//

#ifdef __APPLE__  // include Mac OS X verions of headers
#  include <OpenGL/OpenGL.h>
#  include <GLUT/glut.h>
#else // non-Mac OS X operating systems
#  include <GL/glew.h>
#  include <freeglut.h>
#  include <freeglut_ext.h>
#endif  // __APPLE__

// Define a helpful macro for handling offsets into buffer objects
#define BUFFER_OFFSET( offset )   ((GLvoid*) (offset))
```

```
//----------------------------------------------------------------------------
//
//  --- Include our class libraries and constants ---
//

namespace Angel {

//  Helper function to load vertex and fragment shader files
GLuint InitShader( const char* vertexShaderFile,
                   const char* fragmentShaderFile );

//  Defined constant for when numbers are too small to be used in the
//    denominator of a division operation.  This is only used if the
//    DEBUG macro is defined.
const GLfloat  DivideByZeroTolerance = GLfloat(1.0e-07);

//  Degrees-to-radians constant
const GLfloat  DegreesToRadians = (GLfloat) M_PI / (GLfloat)180.0;

}  // namespace Angel

#include "vec.h"
#include "mat.h"
#include "CheckError.h"

#define Print(x)  do { std::cerr << #x " = " << (x) << std::endl; } while(0)

//  Globally use our namespace in our example programs.
using namespace Angel;

#endif // __ANGEL_H__
```

C – Hướng dẫn

1. **Kiểm tra thông tin card đồ họa**: phiên bản OpenGL?

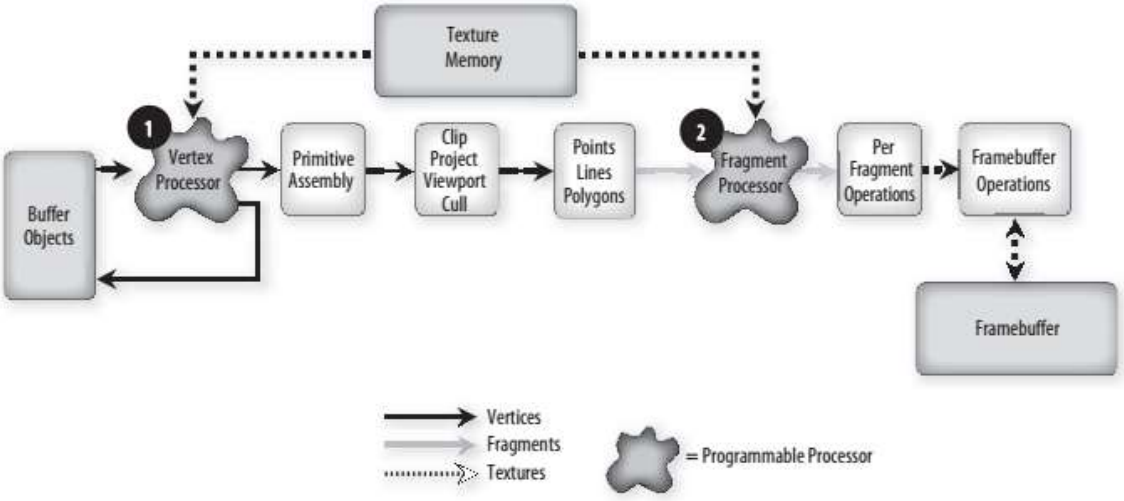
Tải và chạy phần mềm *opengl-extensions-viewer* để xem phiên bản OpenGL của máy.

⇨ Nếu phiên bản OpenGL của máy từ 4.0 trở lên thì phù hợp với khai báo trong project trên, còn thấp hơn sẽ khai báo lại trong **vshader1.glsl** và **fshader1.glsl** tương ứng như bảng dưới về Shader preprocessor.

Các phiên bản tương ứng của glsl với openGL và khai báo trong shader:

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59[1]	2.0	30 April 2004	#version 110
1.20.8[2]	2.1	07 September 2006	#version 120
1.30.10[3]	3.0	22 November 2009	#version 130
1.40.08[4]	3.1	22 November 2009	#version 140
1.50.11[5]	3.2	04 December 2009	#version 150
3.30.6[6]	3.3	11 March 2010	#version 330
4.00.9[7]	4.0	24 July 2010	#version 400
4.10.6[8]	4.1	24 July 2010	#version 410
4.20.11[9]	4.2	12 December 2011	#version 420
4.30.8[10]	4.3	7 February 2013	#version 430
4.40.9[11]	4.4	16 June 2014	#version 440
4.50.7[12]	4.5	09 May 2017	#version 450
4.60.5[13]	4.6	14 June 2018	#version 460

2. Chu trình xử lý đồ họa OpenGL khả lập trình (Theo Chương 4, Orange book)

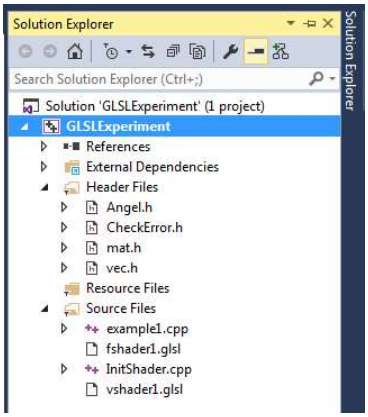


- Để kết xuất một hình ảnh bắt buộc lập trình viên phải cài đặt 2 shader: Vertex shader và Fragment shader (tương ứng với Vertex Processor và Fragment Processor). Việc cài đặt các shader phải sử dụng ngôn ngữ GLSL – Graphic Library Shader Language, là một bộ phận của OpenGL.

- Thông thường để cài đặt Vertex shader và Fragment shader được tổ chức thành 2 file. Do vậy cấu trúc một chương trình OpenGL hiện đại thường có 3 file:
  - 1 File *.cpp: chứa các lệnh OpenGL, file chính để tạo ứng dụng
  - 1 File để cài đặt Vertex Shader
  - 1 File để cài đặt Fragment Shader

Khi file *.cpp có kích thước lớn và có các hàm tái sử dụng trong các ứng dụng OpenGL khác nhau thì ta sẽ tách thành các file thư viện như các file *.h (tương ứng có các file *.cpp cài đặt của các file *.h).

Ví dụ: trong project mục A ta có:



- 2 file cài đặt shader: fshader1.glsl và vshader1.glsl (có thể soạn thảo trên VS C++ hoặc Node pad, dạng file text)

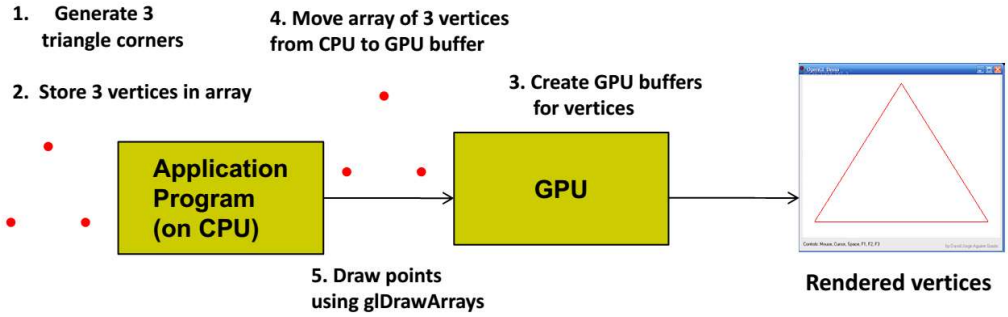
- File .cpp chính của project: example1.cpp

- File thư viện tự xây dựng được sử dụng trong ứng dụng: Angel.h và cài đặt của nó là InitShader.cpp.

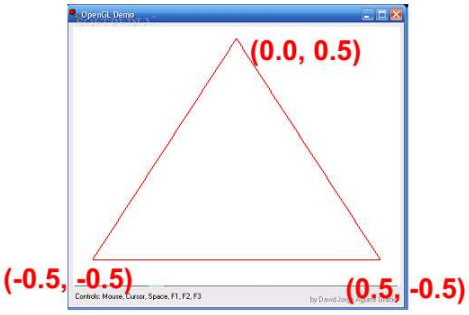
3. Các bước lập trình với kiến trúc OpenGL hiện đại

// Trình bày thông qua Project thuộc mục A.

- **Các bước để kết xuất ra hình tam giác**:
  1. Tính 3 đỉnh để tạo tam giác
  2. Lưu 3 đỉnh vào một mảng
  3. Tạo GPU buffer cho các đỉnh
  4. Di chuyển mảng 3 đỉnh từ CPU đến GPU buffer
  5. Vẽ 3 điểm từ mảng trên GPU sử dụng glDrawArray



- **Bước 1, 2: Tính 3 đỉnh và lưu vào mảng các đỉnh**

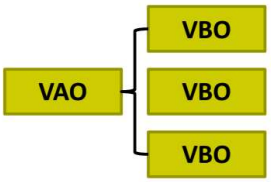



```
int main( int argc, char **argv )
{
   // main function: program starts here

    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB
    glutInitWindowSize( 640, 480 );
    glutInitWindowPosition(100,150);
    glutCreateWindow( "Starting App" );

    glewInit();        /* Khởi tạo glew */

    generateGeometry( );  /* Gọi hàm sinh các đi
    initGPUBuffers( );    /* Tạo GPU buffers*/
    shaderSetup( );       /* Kết nối file .cpp hiện tại với các file shader */

    glutDisplayFunc( display );    // Đăng ký hàm gọi lại về hiển thị
    glutKeyboardFunc( keyboard );  // Đăng ký hàm gọi lại cho sự kiện bàn phím

        /* Có thể thêm các hàm gọi lại về điều khiển chuột, thay đổi kích cỡ cửa sổ, ... ở đây*/

    glutMainLoop();   // Đưa vào vòng lặp vô hạn
    return 0;
}
```

```
typedef vec2 point2;
// Số các đỉnh của tập các đoạn thẳng
const int NumPoints = 3;
// Mảng các đỉnh của hình cần vẽ
point2 points[NumPoints];

void generateGeometry( void )
{
        // Gán giá trị các đỉnh cho mảng
        points[0] = point2(-0.5, -0.5 );
        points[1] = point2( 0.0, 0.5 );
        points[2] = point2( 0.5, -0.5 );
}
```

- **Bước 3. Tạo GPU buffer cho các đỉnh (VAOs, VBOs)**

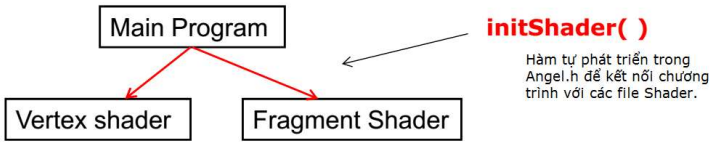
```
void main(int argc, char** argv){
   glutInit(&argc, argv);   // initialize toolkit
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(640, 480);
   glutInitWindowPosition(100, 150);
   glutCreateWindow("my first attempt");
   glewInit( );

   // … now register callback functions
   glutDisplayFunc(myDisplay);
   glutReshapeFunc(myReshape);
   glutMouseFunc(myMouse);
   glutKeyboardFunc(myKeyboard);

   glewInit( );
   generateGeometry( );
   initGPUBuffers( );

   glutMainLoop( );
}
```



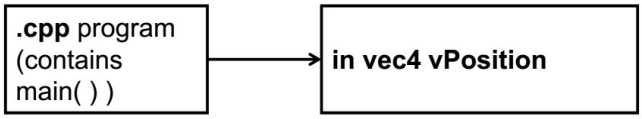

```
void initGPUBuffers( void )
{
   // Create a vertex array object
   GLuint vao;
   glGenVertexArrays( 1, &vao );
   glBindVertexArray( vao );

   // Create and initialize a buffer object
   GLuint buffer;
   glGenBuffers( 1, &buffer );
   glBindBuffer( GL_ARRAY_BUFFER, buffer );
   glBufferData( GL_ARRAY_BUFFER,
         sizeof(points), points, GL_STATIC_DRAW );
}
```

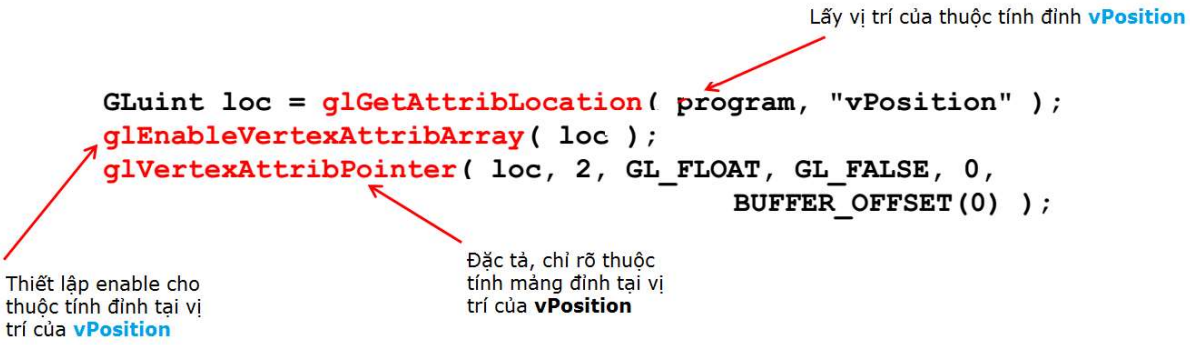
- **Bước 4. Di chuyển mảng 3 đỉnh từ CPU đến GPU buffer**
  - Các file và hàm thực hiện
    - File vshader1.glsl: chứa các cài đặt cho vertex shader
    - File fshader1.glsl: chứa các cài đặt cho fragment shader
    - File Angel.h, Initshader.cpp: hàm InitShader
    - File example1.cpp: hàm **setupShader**
  - Hàm **setupShader** thực hiện kết nối chương trình với các shader. Cụ thể:
    - Tạo biến quản lý chương trình – program, kết nối với 2 file shader và khai báo sử dụng đối tượng program để thực hiện xử lý các đỉnh trên vertex shader và fragment đưa vào.
      - Lưu ý: Hàm thực hiện chủ yếu là hàm InitShader trong Angel.h và Initshader.cpp

```
GLuint = program;
GLuint program = InitShader( "vshader1.glsl", fshader1.glsl");
glUseProgram(program);
```

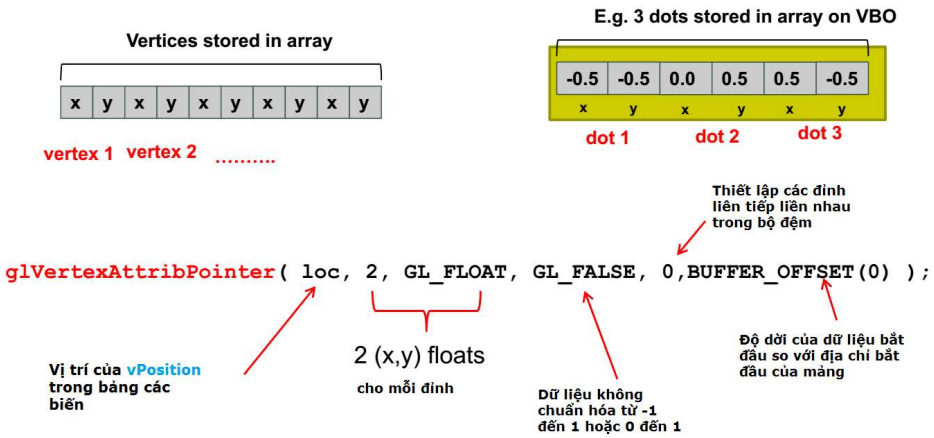


- Thiết lập các thuộc tính đỉnh - Vertex Attributes
  - Nhằm kết nối các đỉnh đang lưu trong VBO với biến input (khai báo: in) của shader (biến vPosition đã khai báo trong vertex shader) để xử lý.





  - glVertexAttribPointer



- Vị trí của hàm setupShader trong chương trình chính

```
void main(int argc, char** argv){
   glutInit(&argc, argv);   // initialize toolkit
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(640, 480);
   glutInitWindowPosition(100, 150);
   glutCreateWindow("my first attempt");
   glewInit( );

   // … now register callback functions
   glutDisplayFunc(myDisplay);
   glutReshapeFunc(myReshape);
   glutMouseFunc(myMouse);
   glutKeyboardFunc(myKeyboard);

   glewInit( );
   generateGeometry( );
   initGPUBuffers( );
   void shaderSetup( );

   glutMainLoop( );
}
```

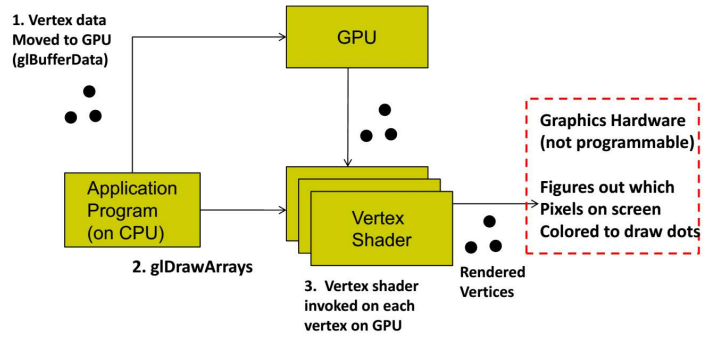
```
void shaderSetup( void )
{
   // Load shaders and use the resulting shader program
   program = InitShader( "vshader1.glsl", "fshader1.glsl" );
   glUseProgram( program );

   // Initialize vertex position attribute from vertex shader
   GLuint loc = glGetAttribLocation( program, "vPosition" );
   glEnableVertexAttribArray( loc );
   glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0,
                                                BUFFER_OFFSET(0) );

   // sets white as color used to clear screen
   glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```

- Vertex shader
  - Biến gl_Position là biến được xây dựng sẵn trong Graphic Pipeline
  - Các biến tự khai báo trong shader:
    - in: biến đầu vào
    - out: biến đầu ra

```
in vec4 vPosition

void main( )
{
      gl_Position = vPosition;
}
```

output vertex position — input vertex position

  - Mô hình thi hành

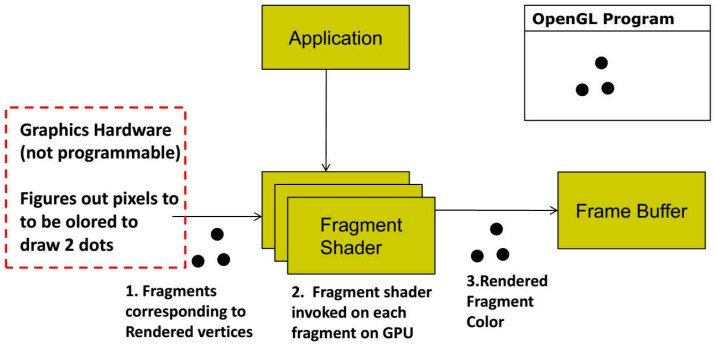


- Fragment shader
  - Biến gl_FragColor là biến được xây dựng sẵn trong Graphic Pipeline

```
void main( )
{
      gl_FragColor = vec(1.0, 0.0, 0.0, 1.0);
}
```

Set each drawn fragment color to red

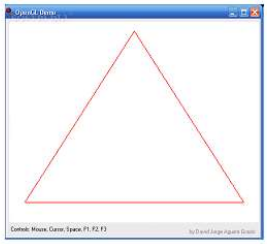
  - Mô hình thi hành



- **Bước 5. Vẽ 3 điểm từ mảng trên GPU sử dụng glDrawArray**

```
glDrawArrays(GL_POINTS, 0, N);
```

Render buffered data as points | **Starting index** | **Number of points to be rendered**

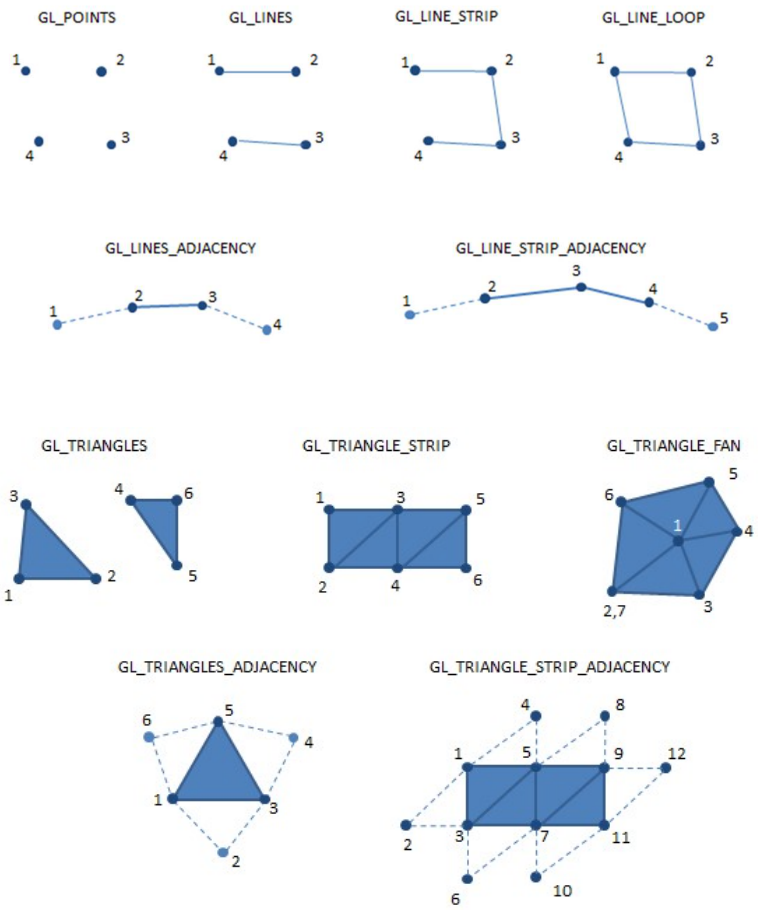


- Display function using **glDrawArrays**:

```
void mydisplay(void){
    glClear(GL_COLOR_BUFFER_BIT);     // clear screen
    glDrawArrays(GL_LINE_LOOP, 0, 3);   // draw the points
    glFlush( );                  // force rendering to show
}
```

Một số tham số hình vẽ của glDrawArrays:

glDraw command	output primtive	used in shaders
GL_POINTS	points	geometry; fragment
GL_LINES	lines	geometry; fragment
GL_LINE_STRIP	lines	geometry; fragment
GL_LINE_LOOP	lines	geometry; fragment
GL_LINES_ADJACENCY	lines_adjacency	geometry
GL_LINE_STRIP_ADJACENCY	lines_adjacency	geometry
GL_TRIANGLES	triangles	geometry; fragment
GL_TRIANGLE_STRIP	triangles	geometry; fragment
GL_TRIANGLE_FAN	triangles	geometry; fragment
GL_TRIANGLES_ADJACENCY	triangles_adjacency	geometry
GL_TRIANGLE_STRIP_ADJACENCY	triangles_adjacency	geometry
GL_PATCHES	patches	tessellation control



4. Các hàm trong chương trình

`GLuint` **`glCreateProgram`**`(void)`
Ví dụ:
`GLuint shader = glCreateProgram();`
	Tạo ra một đối tượng program

`GLuint` **`glCreateShader`**`(GLenum` *`shaderType`*`)`
 	
Trong đó: *`shaderType`*
– `GL_VERTEX_SHADER`
– `GL_FRAGMENT_SHADER`
– …
Ví dụ:
`GLuint shader1 = glCreateShader(GL_VERTEX_SHADER);`
`GLuint shader2 = glCreateShader(GL_FRAGMENT_SHADER);`
	Tạo ra một đối tượng shader theo loại shader được truyền vào

`void` **`glShaderSource`**`(GLuint` *`shader`*`, GLsizei` *`count`*`, const GLchar **`string`*`, const GLint *`length`*`)`
 	
Trong đó:
 	
`Shader:` Đối tượng shader được tạo bởi lệnh `glCreateShader`.
 	
`Count:` Số phần tử của mảng các `string` hoặc `length` tương ứng.
 	
`String:` Mảng các con trỏ chỉ tới các chuỗi chứa mã nguồn được nạp vào shader.
 	
`Length:` Mảng của các độ dài chuỗi trên (*`string`*).
Nếu length là NULL thì mỗi chuỗi phải được kết thúc bằng giá trị null (‘\0’)
Nếu độ dài là một giá trị khác với NULL, thì nó trỏ đến một mảng chứa độ dài chuỗi cho mỗi phần tử tương ứng của chuỗi.
Ví dụ:
`glShaderSource(shader1, 1, (const GLchar**) &s.source, NULL);`
`// s.source kiểu GLchar*, trỏ đến biến chứa mã nguồn của shader1.`
Nghĩa là: shader1 được gán giá trị mã nguồn được xác định bởi 1 xâu và được chỉ ra ở &s.source và xâu này của mã nguồn được kết thúc bằng kí tự null – ‘\0’
	Gán mã nguồn (code) trong đối tượng *`shader`*

`void` **`glCompileShader`**`(GLuint` *`shader`*`)` Lưu ý: Thông tin về dịch shader thành công hay không thành công được gọi bởi hàm `glGetShaderInfoLog`.	Dịch đối tượng `shader`.
`void` **`glLinkProgram`**`(GLuint program)`	Liên kết đối tượng shader
…	

D – Bài tập

Lưu ý: Toàn bộ các bài tập được sửa từ bài mẫu trên, không cần code lại từ đầu.

1. Vẽ hình tam giác, hình chữ nhật, hình quạt được tô kín hình trong không gian 2D.
2. Vẽ hình lập phương có cạnh =1, tâm O(0, 0, 0) ở chính giữa hình lập phương. [1]