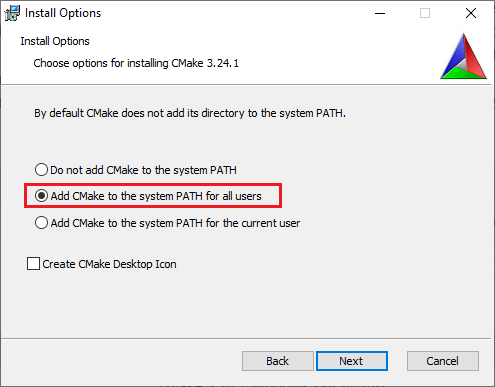
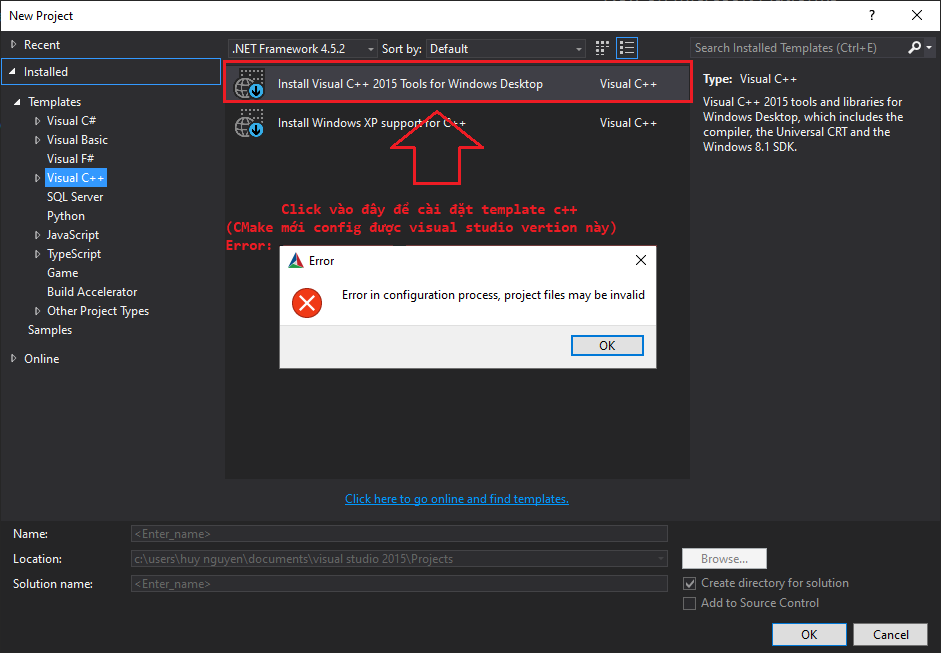
1. Mở đầu
   1. Yêu cầu phần mềm
      1. Visual studio 2015
         * Bắt buộc phải có template c++ khi tạo ra 1 project mới(nếu không có thì cmake không thể nào config được)
      2. Cmake (Sử dụng thư viện c++ bên ngoài – Visual 2015)
         * Trang download Cmake: <https://cmake.org/download/>
         * Chú ý khi cài đặt Cmake phải chọn add path…



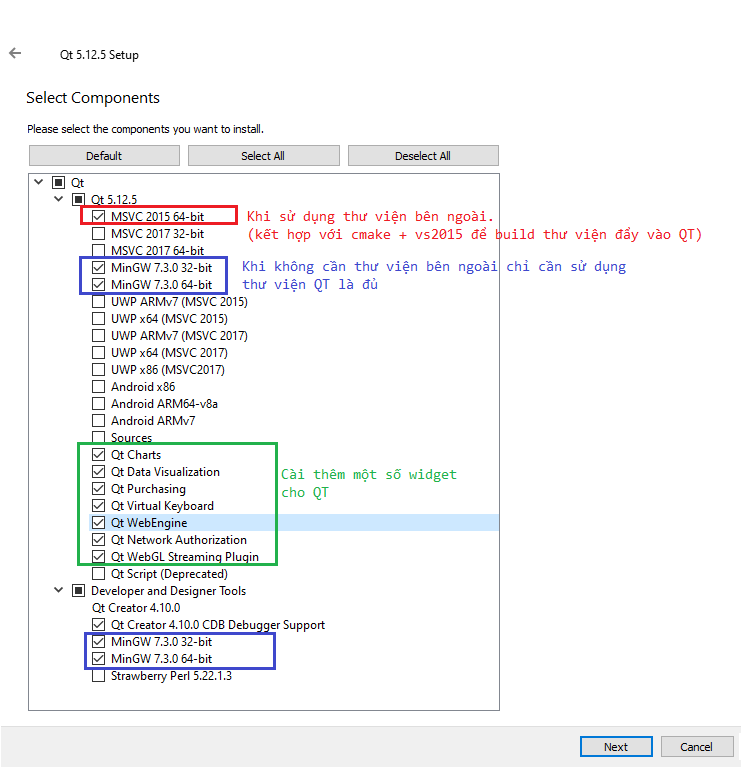
* + - * Khi config một thư viện bất kỳ chú ý 2 điểm sau:
        + Phải cài đặt visual studio đó trên máy
        + Khi tạo project mới lựa chọn c++ nếu có template c++ tức là config trong Cmake là thành công.Ngược lại nếu không có template c++ thì người dùng bắt buộc phải cài đặt template này.



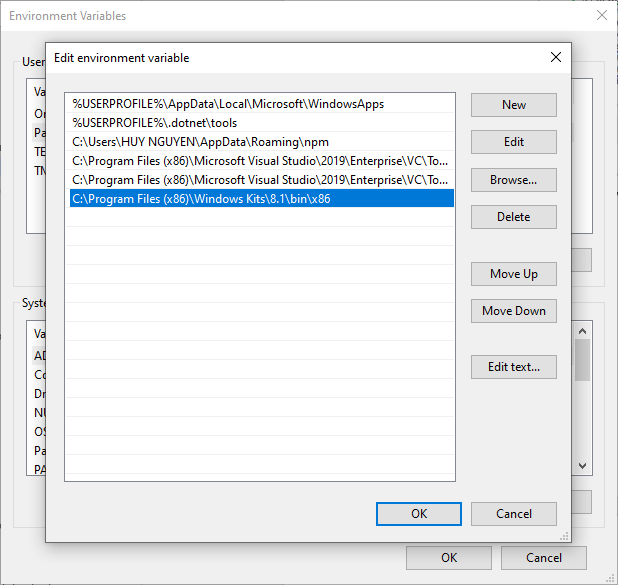
* + 1. QT phiên bản từ 5.9
       - Trang download QT creator: <https://download.qt.io/archive/qt/>



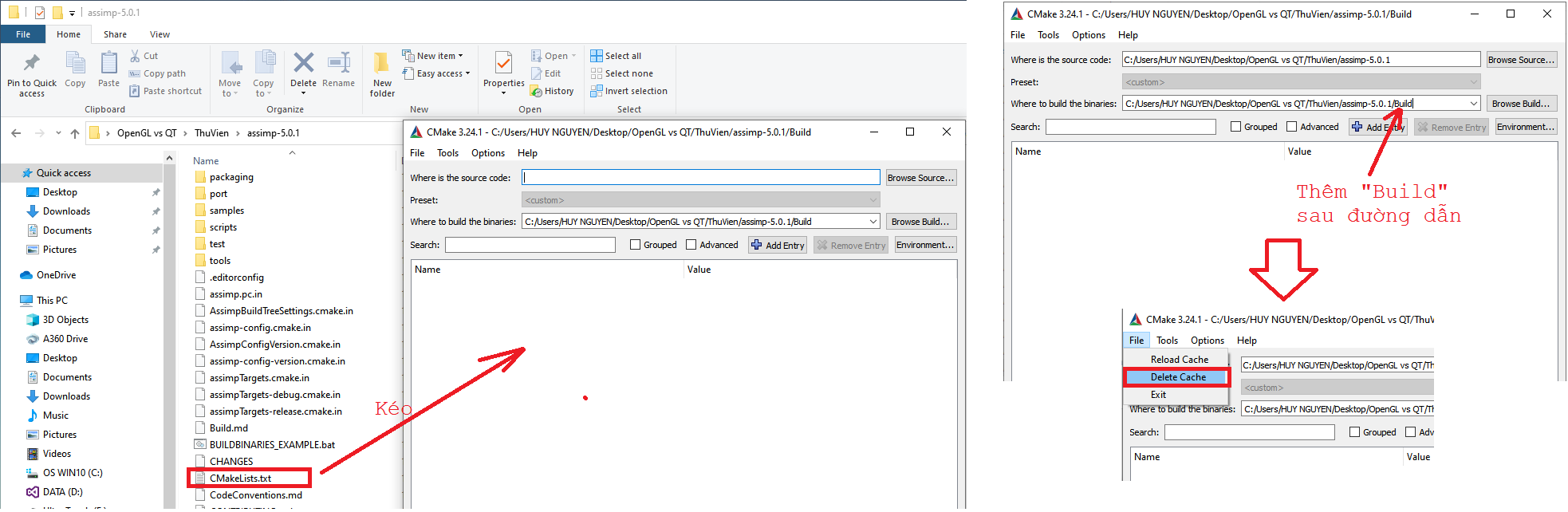
* + - * Chú ý khi cài đặt QT chọn cài đặt theo các lựa chọn sau



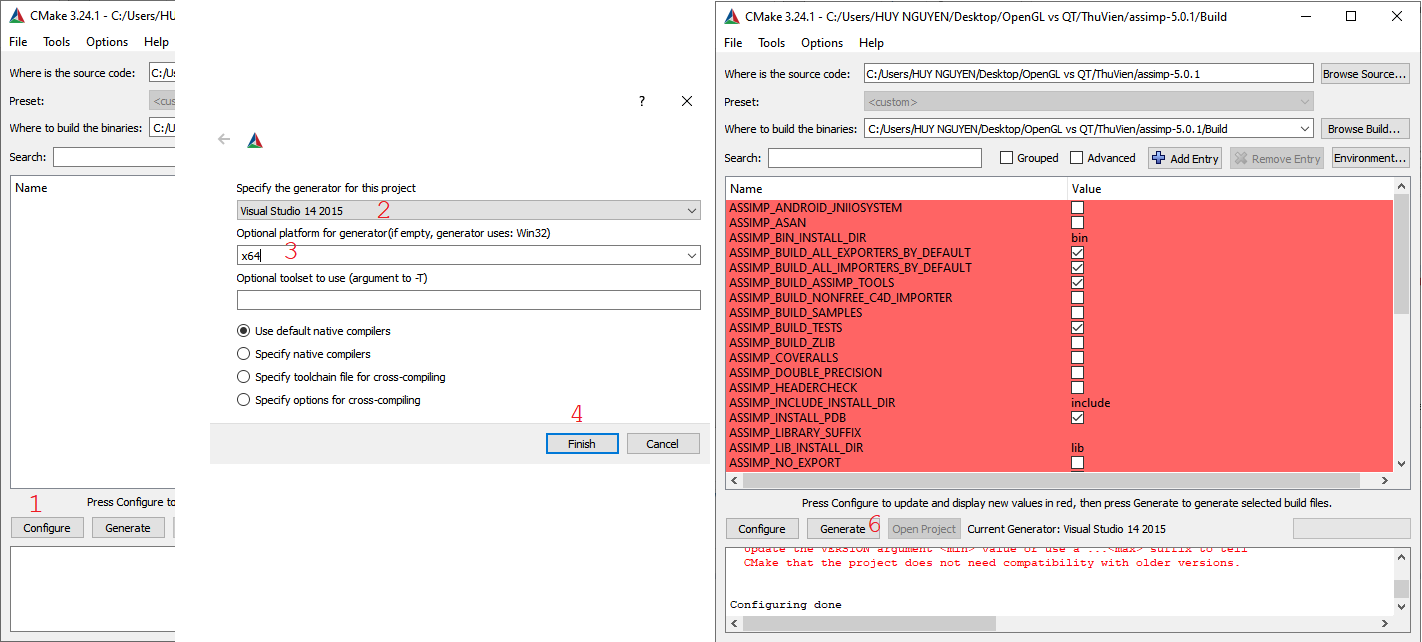
* + - * Lỗi Link1158:cannot run “rc.exe”. Ta chỉ cần thêm đường dẫn sau “C:\Program Files (x86)\Windows Kits\8.1\bin\x86” vào enviroment là được(nếu không được cài window deverlop kits)



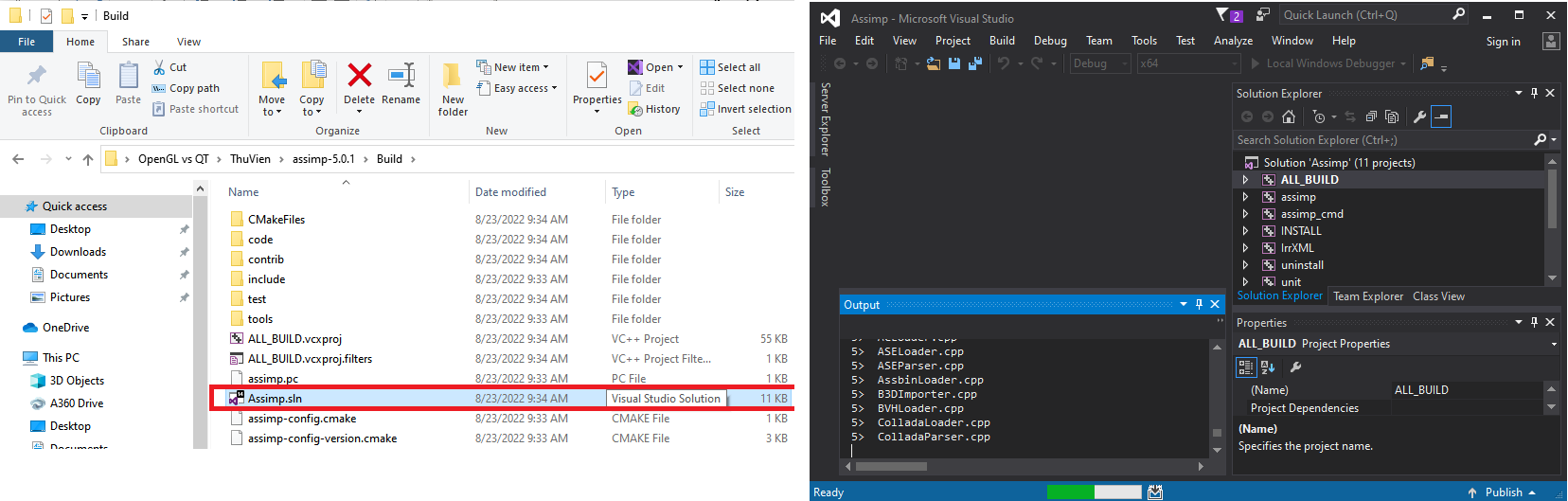
* 1. Tài liệu tham khảo
     1. Link tra api: <https://docs.gl/gl3/glCheckFramebufferStatus>
     2. Link học: <https://learnopengl.com/>
  2. Thao tác cài đặt và sử dụng thư viện bên ngoài
     1. Assimp(Assimp: 5.0.1 – QT: 5.9 – MSCV2015)
        + Bước 1: tải thư viện assimp về máy sau đó giải nén
        + Bước 2: Mở CMake lên và kéo file txt vào Cmake.



* + - * Bước 3: config sau đó nhấn generate

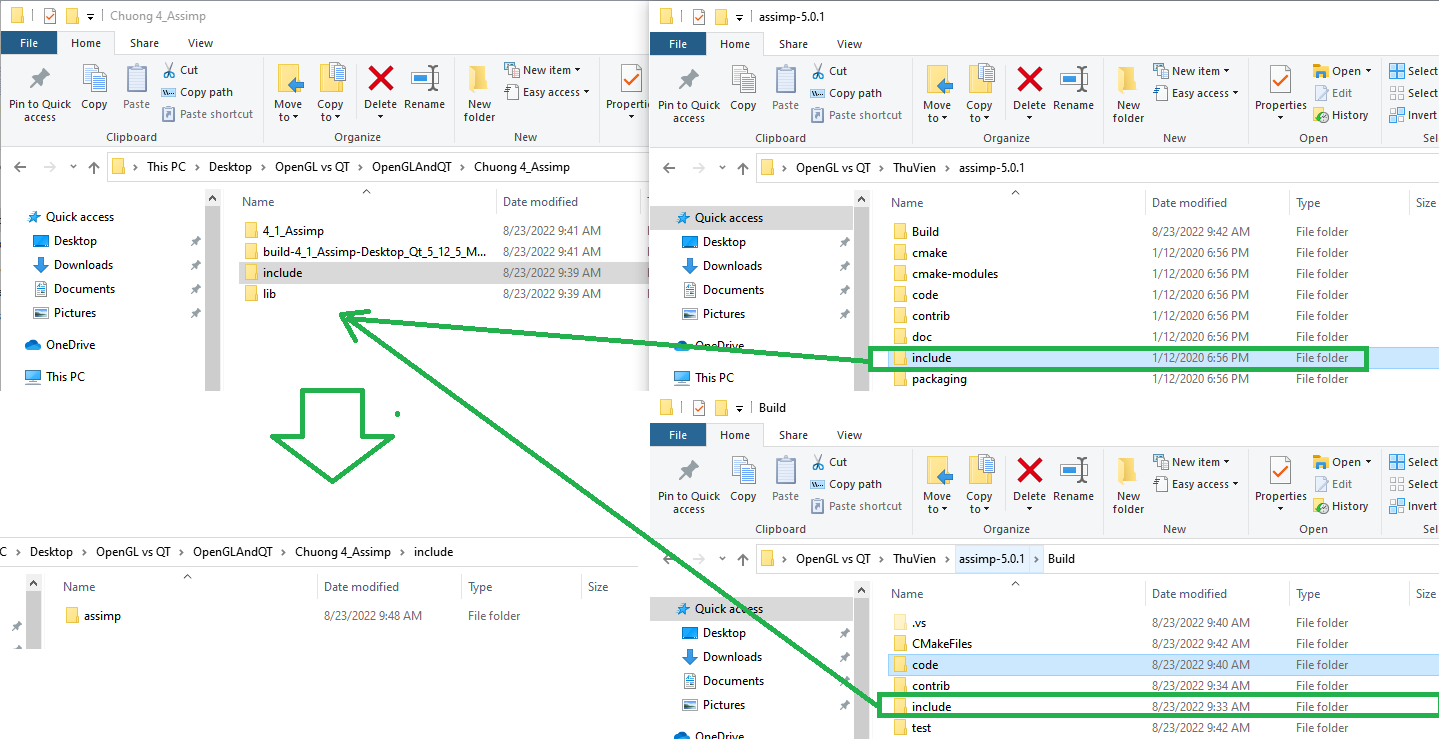


* + - * Bước 4: mở file .sln lên và build với 2 phiên bản debug và release

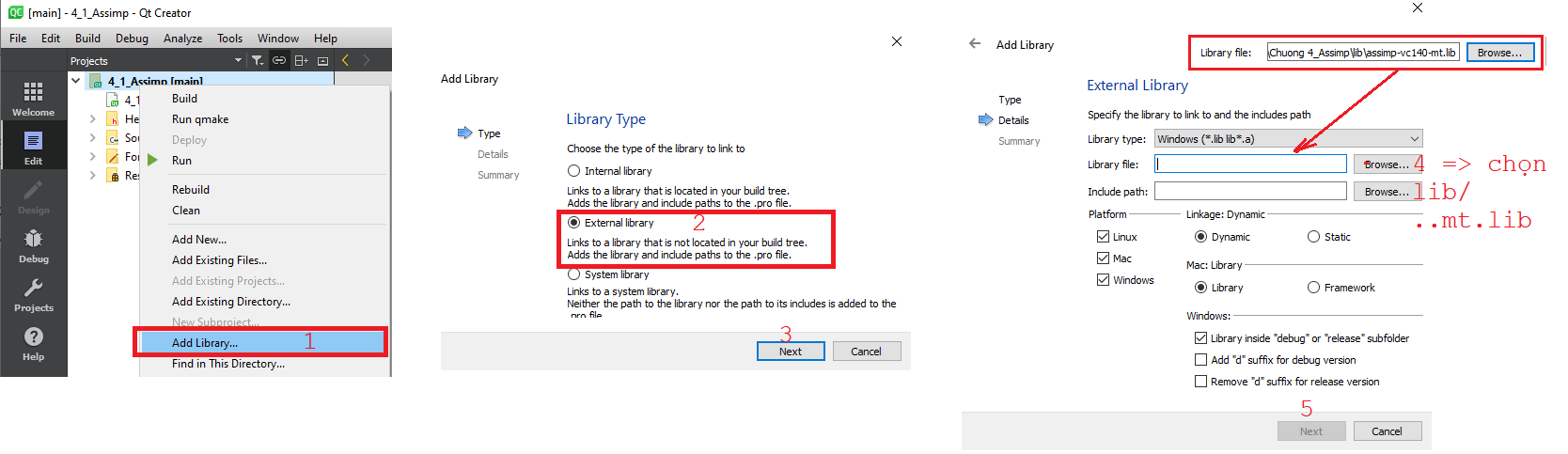


* + - * Bước 5: Tạo 2 folder “include” và “lib” trong project của mình và copy thư viện đã build vào 2 thư mục này



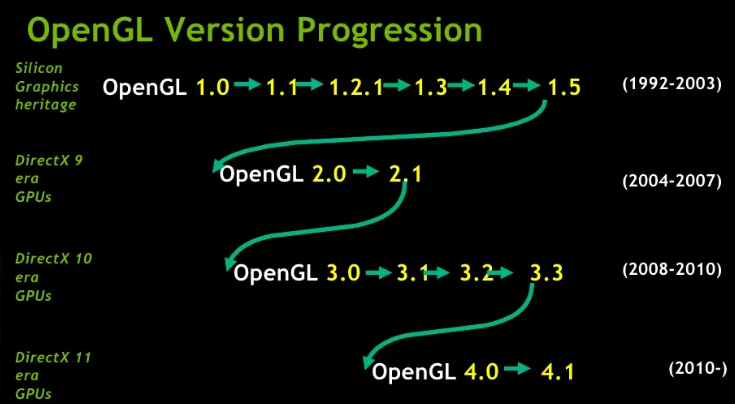


* + - * Bước 6: Mở project và add thư viện

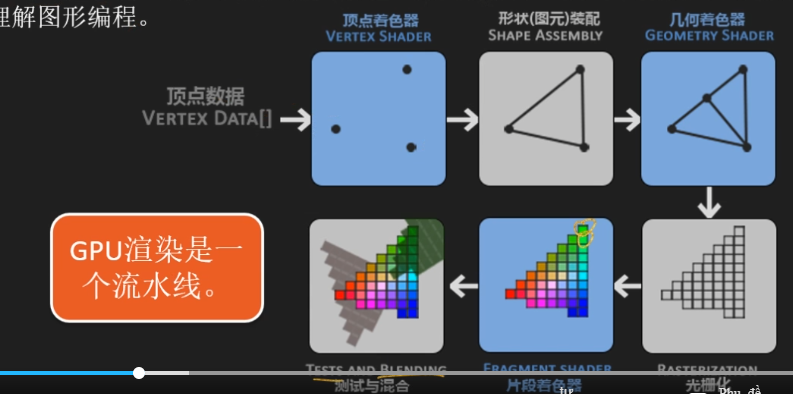


Như vậy là đã add thư viện thành công

1. Tìm hiểu opengl
   1. OpenGL và QT
      * + Sử dụng OpenGL trong QT sẽ giúp chúng ta tiết kiệm rất nhiều thời gian viết code.Một số chức năng của opengl phải sử dụng nhiều đoạn code mới làm được nhưng đối với QT nó đã đóng sẵn thành class để ta dễ dàng thao tác.Ví dụ: QOpenGLShaderProgram, QopenglTexture,…
        + Opengl được viết ở rất nhiều ngôn ngữ khác nhau nhưng chung về nguyên tắc, cú pháp và hiệu xuất.
        + Phiên bản 3.2 (Mô hình truyền thống) và phiên bản 3.3 được gọi mà mô hình hiện đại.
        + Các card đồ họa khác nhau được sản xuất bởi các nhà sản xuất khác nhau. Trình điều khiển crard đồ họa của Apple là gì?Nó được tạo bởi apple. Nhưng nền tảng khác nhau thì sao ? Các trình điều khiển của card đồ họa được thực hiện bởi các nhà sản xuất khác nhau. Bản thân OpenGL không phải là một tập các API đó là một tiêu chuẩn, một tiêu chuẩn của riêng openGL bạn chỉ cần áp dụng đúng tiêu chuẩn này.Để gọi một chức năng trong card đồ họa chúng ta cần trỏ tới nó nhưng địa chỉ của chức năng cụ thể của trình điều khiển cạc đồ họa chỉ được biết sau khi chạy. Chức năng này của mỗi card đồ họa nằm trên “Bộ nhớ” riêng của nhà sản xuất. Sau đó tôi gọi đến tôi có quá nhiều chức năng và tôi sẽ tìm từng chức năng một quá trình này phức tạp.
        + Trình biên dịch MinGW và trình biên dịch MSVC
          - MninGW: chỉ áp dụng khi không có thư viện dùng bên ngoài.
          - MSCV: sử dụng thư viện dùng bên ngoài. Dùng Cmake để build ra visual studio . Sau đó mở Visual studio build/release lại sau đó đẩy vào QT. thì QT sẽ được sử dụng thư viện đó. Để chạy được thư viện này bắt buộc phải dùng trình biên dịch MSCV.
   2. Sự khác nhau giữ MFC và QT
      * + MFC chỉ chạy được trên môi trường Window.
        + QT chạy được đa nền tảng.
   3. OpenGL là gì
      1. Thế nào là openGl
         * OpenGL : Open Graphics Library, đây là nột thông số kỹ thuật được phát triền và duy trì bởi tổ chức Khronos.Thông số tiêu chuẩn được hiểu nó như một số tài liệu một số chức năng cụ thể của bạn đầu vào và đầu ra của nó là gì.Làm thế nào để làm điều đó trong nội bộ.Do đó mỗi nhà sản xuất có một mức độ tự chủ nhất định cái đó dành cho các lập trình viên.
         * Khi nói về OpenGL họ thường nghĩ viết code bằng C.Nhưng thực chất OpenGL có thể viết bằng nhiều ngôn ngưc khác nhau.
      2. Lịch sử phát triển của opengl



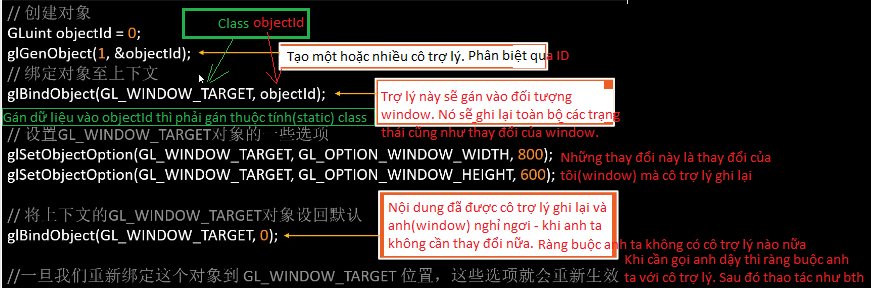
* + 1. Opengl(Core profile)
       - Core profile: được gọi là chế độ hiện đại , được gọi là chế độ có thể lập trình (pipeline). Sự khác biệt giữa Vertex shader,geometry shader,fragment shader.
         * Geometry shader: Là tự động, người dùng có thể bật tắt tính tự động này => không thể can thiệp được code.
         * Chúng ta tự viết chương trình cho vertexshader và fragmentshader.Sau đó GPU sẽ thực thi và liên kết 6 phần này với GPU trước khi có thể được sử dụng.
       - Hệ thống pipeline
         * Vertex shader: phần này dùng để xác định vị trí đỉnh.
         * Shape Assembly:Từ các vị trí này sẽ tập hợp thành một nguyên thủy(pri…) cụ thể.
         * Geometry Shader: Bên trong các nguyên thủy ta có thêm, sửa,xóa, cập nhật,… các tọa độ điểm mới hoặc đã có.
         * Rasterization:Biến đổi các nguyên thủy thành các pixel nhỏ.
         * Fragment Shader:Gán màu vào từng đỉnh và các pixel sẽ tự nội suy màu.
         * Test and Blending: Test và hợp nhất giữa các pixel.
       - Chú ý
         * Bản opengl 3.2 trở xuống thì người lập trình không thể can thiệp được vào Shape Assembly,Test and blending, Rasterization.(Xám) mà opengl đã viết sẵn rồi => pipeline cố định.



* 1. Chế độ hiển thị ngay lập tức
     + - Hệ thống pipeline đã nói ở trên được gọi là kiết suất cố định.
       - Những gì hầu hết các chức năng Opengl tất cả bị che dấu nó dễ sử dụng và dễ hiểu nhưng kém hiệu quả.
       - Nhà phát triển có rất ít quyền kiểm soát đối với cách tính số lần hủy mở
       - Bản openGL 3.2 bắt đầu có chế độ mới.
  2. State machine
     + - OpenGL chính là một cỗ máy. Một bộ tập hợp tất cả các biến mô tả làm thế nào để hoạt động.
       - OpenGL liên quan đến Contex.
       - Cùng một đoạn code nhưng sẽ có 2 trạng thái khác nhau: glbind..,… => kiểm xoát trạng thái của opengl
         * Trạng thái thay đổi: State-changing function.(thay đổi biến – để set giá trị vào biến đó)
         * Trạng thái sử dụng: State-using Function.
       - Hiểu đơn giản state machine chính là thay đổi 1 số biến mà kết quả đầu ra sẽ dựa theo những biến đó.
  3. Object trong openGL
     + - Mỗi một đối tượng đề cập đến một số các state(biến) trong state machine.Chúng ta biến rằng opengl là một state machine lớn.Chứa nhiều con trỏ trong máy trạng thái này
       - Ví dụ: Ta tạo 1 đối tượng(Tạo ra một cấu trúc(Object\_Name)) có chứa các tùy chọn như màu, kích thước,…Và trong mỗi tùy chọn của Object\_name lại chứa các tùy chọn con của nó nữa,… (tree)

|  |
| --- |
| Struct object\_name{  GLfloat option1;  GLunit option2;  GLchar[] namel  }  Struct Open\_Context{  …  Object\* object\_Window\_Target;  …  } |

* + - * Ví dụ code



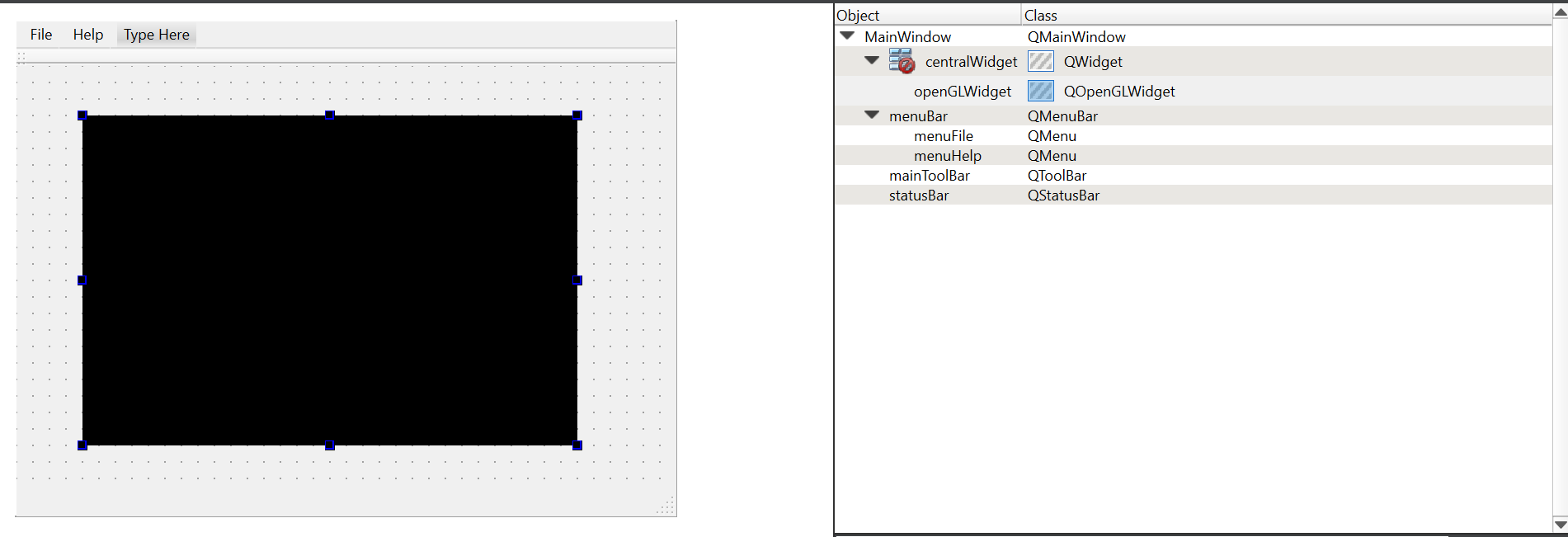
1. Liên kết OpenGL và QT
   1. Hello widget
      1. Lý thuyết
         * QOpenGLWidget
           + Là một lớp cung cấp cửa sổ window và có thể viết mã opengl. Mã opengl viết được trong các phương thức sau:

paintGL

resizeGL

initiaizeGL: Khởi tạo ban đầu chạy trước paintGL và resizeGL.

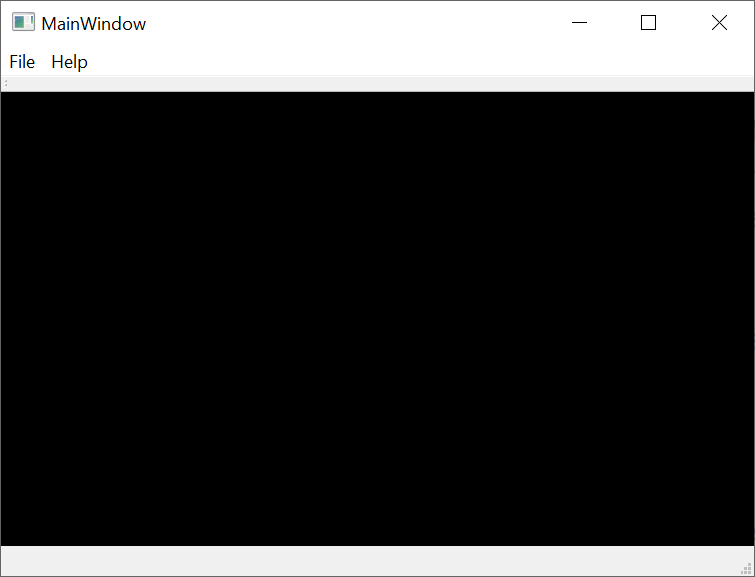
* + - * + Phương thức update() sử dụng khi nằm ngoài phương thức paintGL() có chức năng giống repaint lại cửa sổ.
        + Context sử dụng code opengl sẽ nằm trong các phương thức paintGL(), resizeGL(), initializeGL(). Nếu muốn sử dụng code opengl nằm ngoài phương thức này thì ta cần dùng phương thức makeCurrent().
      * QOpenGLFunctions\_x\_x\_Core
        + Sử dụng con trỏ chức năng để gọi một chức năng của card đồ họa.Hàm này được gọi thông qua mã opengl.
      * Khi khởi tạo QOpenGLWidget cần phải khởi tạo đối tượng này thông qua phương thức initializeOpenGLFunctions(). Nếu không khởi tạo đoạn code này thì các phương thức opengl sẽ không liên kết với chức năng thực tế trên card đồ họa. Dẫn đến code opengl sẽ không hoạt động được.Sau khi được liên kết thì gọi phương thức của opengl nó sẽ trỏ đến chức năng thực trong card đồ họa.
    1. Thực hành
       - Bước 1:Tạo 1 project mainwindow thông thường
       - Bước 2: thiết kế giao diện



* + - * Bước 3: vào code chỉnh OpenglWidget vào đúng vùng cental

|  |
| --- |
| #include "mainwindow.h"  #include "ui\_mainwindow.h"  MainWindow::MainWindow(QWidget \*parent) :  QMainWindow(parent),  ui(new Ui::MainWindow)  {  ui->setupUi(this);  setCentralWidget(ui->openGLWidget);  }  MainWindow::~*MainWindow*()  {  delete ui;  } |

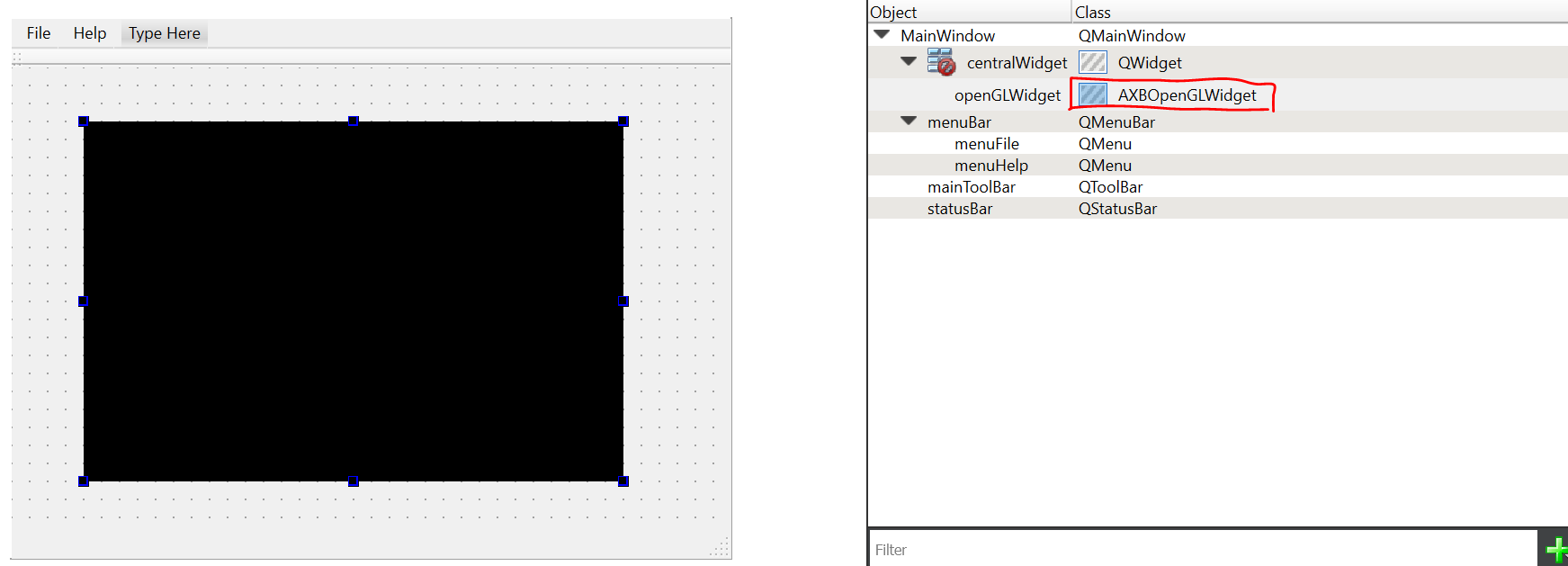
Sau đó chạy nó sẽ hiện ra như sau



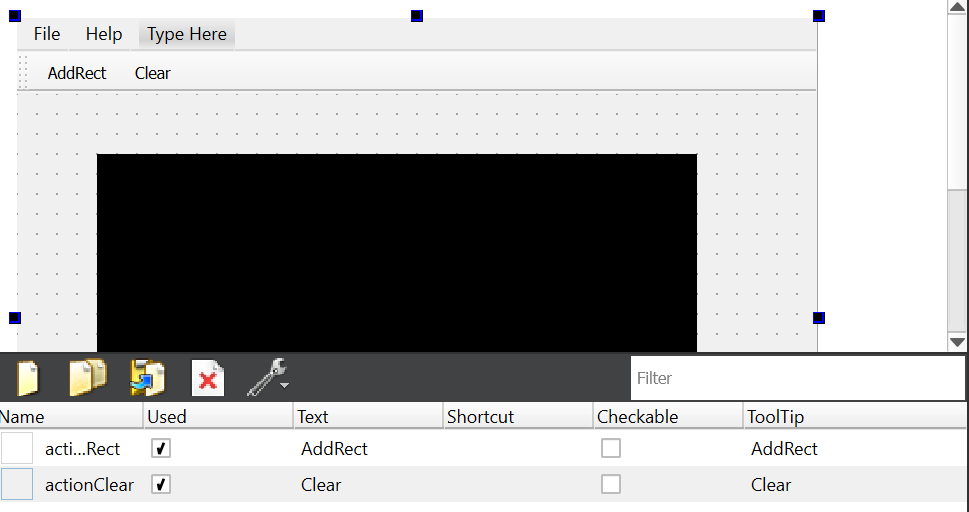
* + - * Bước 4:Tạo mới 1 class AXBOpenGLWidget

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  class AXBOpenGLWidget : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit AXBOpenGLWidget(QWidget \*parent = nullptr);  protected:  void *initializeGL*();  void *resizeGL*(int w, int h);  void *paintGL*();  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  AXBOpenGLWidget::AXBOpenGLWidget(QWidget \*parent) : QOpenGLWidget(parent)  {  }  void AXBOpenGLWidget::*initializeGL*()  {  *initializeOpenGLFunctions*();  }  void AXBOpenGLWidget::*resizeGL*(int w, int h)  {  }  void AXBOpenGLWidget::*paintGL*()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  } |

* + - * Bước 5:Promote OpenglWidget tạo ở bước 2 sang class tạo ở bước 4



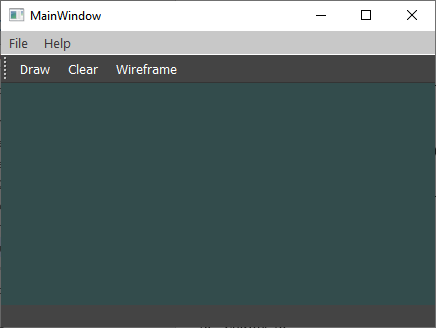
* + - * Bước 6: thêm action vào toolbar



* + - * Bước 7: chạy setstyle window

|  |
| --- |
| QWidget{  background-color: rgb(68,68,68);  color: rgb(255,255,255);  font:10pt;  }  QMenuBar{  background-color: rgb(200,200,200);  color: rgb(60,60,60);  } |

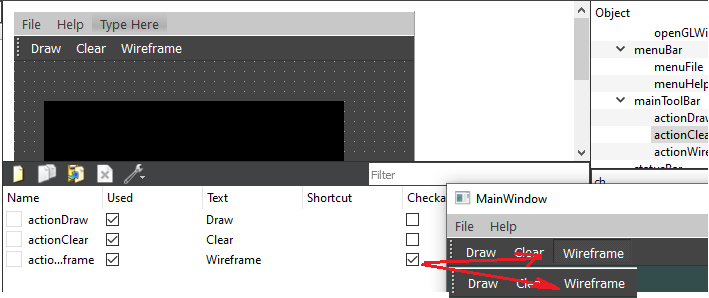
Kết quả



* 1. UI gọi chức năng điều khiển opengl
     1. Lý thuyết
        + Thông thường chúng ta viết mã opengl trong 3 phương thức: paintGL(), resizeGL(), initilizeGL().
        + Nhưng khi ta muốn sử dụng code opengl ở những phương thức khác thì ta cần sử dụng cấu trúc như sau thì opengl mới có hiệu lực

|  |
| --- |
| makeCurrent();  // code opengl  Update();  doneCurrent(); |

* + 1. Thực hành
       - Bước 1: Khởi tạo OpenGL và QT như bình thường
       - Bước 2: sửa giao diện 1 chút



Chỉnh sửa style của window

|  |
| --- |
| QWidget{  background-color: rgb(68,68,68);  color: rgb(255,255,255);  font:10pt;  }  QMenuBar{  background-color: rgb(200,200,200);  color: rgb(60,60,60);  }  QWidget::checked{  background-color: rgb(150,150,150);  } |

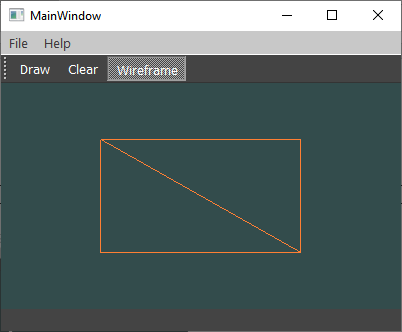
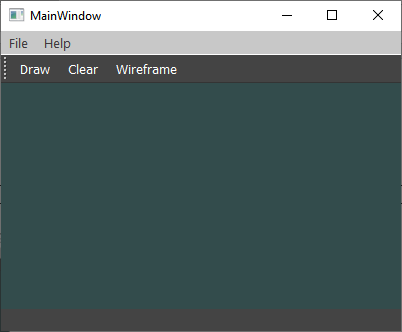
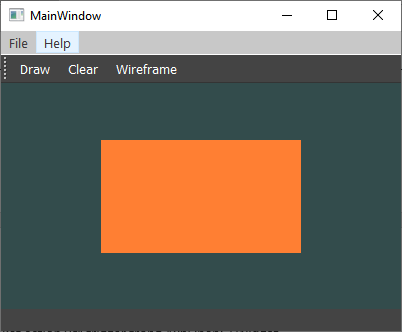
* + - * Bước 3: Thêm các phương thức public trong AxbOpenGLWidget

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include <QWidget>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit **AXBOpenGLWidget**(QWidget \*parent = nullptr);  ~***AXBOpenGLWidget***();  enum **Shape**{None,Rect,Circle,Triangle};  void **drawShape**(Shape shape);  void **setWireframe**(bool wireframe);  protected:  void ***initializeGL***();  void ***resizeGL***(int w, int h);  void ***paintGL***();  private:  unsigned int VAO,VBO,shaderProgram,EBO;  Shape m\_shape;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QDebug>  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget( parent)  {  }  AXBOpenGLWidget::~***AXBOpenGLWidget***()  {  makeCurrent();  glDeleteBuffers(1,&VBO);  glDeleteVertexArrays(1,&VAO);  glDeleteProgram(shaderProgram);  doneCurrent();  }  void AXBOpenGLWidget::**drawShape**(AXBOpenGLWidget::Shape shape)  {  makeCurrent();  m\_shape= shape ;  update();  doneCurrent();  }  void AXBOpenGLWidget::**setWireframe**(bool wireframe)  {  makeCurrent();  if(wireframe)  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);  else  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);  update();  doneCurrent();  }  float vertices[]= {  0.5f, 0.5f,0.0f,  0.5f, -0.5f, 0.0f,  -0.5f, -0.5f, 0.0f,  -0.5f, 0.5f, 0.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  const char \*vertexShaderSource = "#version 330 core\n"  "layout (location = 0) in vec3 aPos;\n"  "void main()\n"  "{\n"  "gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);\n"  "}\0";  const char \*fragmentShaderSource = "#version 330 core\n"  "out vec4 FragColor;\n"  "void main()\n"  "{\n"  "FragColor = vec4(1.0f, 0.5f, 0.2f,1.0f);\n"  "}\0";  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,3\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO) sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  // bước 5\_2: EBO  glGenBuffers(1,&EBO);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof (indices), indices, GL\_STATIC\_DRAW);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);  // ===Vertex Shader =====  unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);  glCompileShader(vertexShader);  int success; char infoLog[512];  glGetShaderiv(vertexShader,GL\_COMPILE\_STATUS, &success);  if(!success){  glGetShaderInfoLog(vertexShader,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog;  }  // == Fagment Shader ===  unsigned int fragShader = glCreateShader(GL\_FRAGMENT\_SHADER);  glShaderSource(fragShader, 1, &fragmentShaderSource, NULL);  glCompileShader(fragShader);  glGetShaderiv(fragShader,GL\_COMPILE\_STATUS, &success);  if(!success){  glGetShaderInfoLog(fragShader,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::FRAGMENT::COMPILATION\_FAILED\n" << infoLog;  }  // == Shader program ==  shaderProgram =glCreateProgram();  glAttachShader(shaderProgram, vertexShader);  glAttachShader(shaderProgram,fragShader );  glLinkProgram(shaderProgram);  glGetShaderiv(shaderProgram,GL\_LINK\_STATUS, &success);  if(!success){  glGetShaderInfoLog(shaderProgram,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::SHADERPROGRAM::COMPILATION\_FAILED\n" << infoLog;  }  // == Xóa Vertex shader, và fragment shader ==  glDeleteShader(vertexShader);  glDeleteShader(fragShader);  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  switch (m\_shape) {  case Rect:  glDrawElements(GL\_TRIANGLES,6,GL\_UNSIGNED\_INT,&indices);  break;  }  } |

* + - * Bước 4:Liên kết action với trigger trong AxbOpenGLWidget

|  |
| --- |
| #include "mainwindow.h"  #include "ui\_mainwindow.h"  MainWindow::**MainWindow**(QWidget \*parent) :  QMainWindow(parent),  ui(new Ui::MainWindow)  {  ui->setupUi(this);  setCentralWidget(ui->openGLWidget);  }  MainWindow::~***MainWindow***()  {  delete ui;  }  void MainWindow::**on\_actionDraw\_triggered**()  {  ui->openGLWidget->drawShape(AXBOpenGLWidget::Rect);  }  void MainWindow::**on\_actionClear\_triggered**()  {  ui->openGLWidget->drawShape(AXBOpenGLWidget::None);  }  void MainWindow::**on\_actionWireframe\_triggered**()  {  ui->openGLWidget->setWireframe(ui->actionWireframe->isChecked());  } |

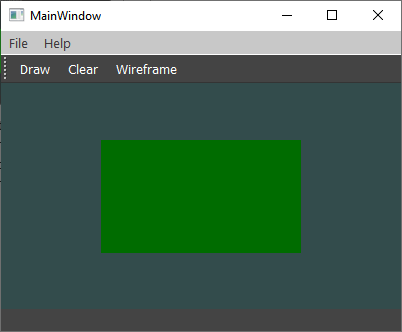
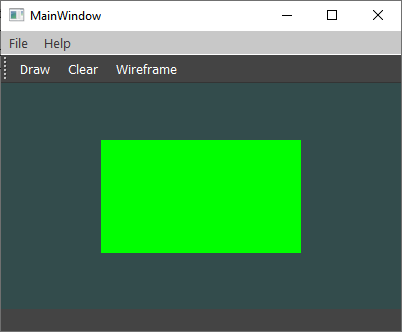
Kết quả



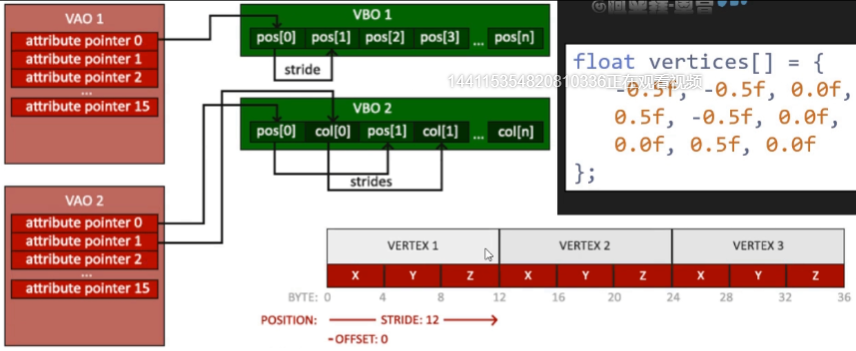
* 1. Tạo animation(vòng lặp while như trong opengl bình thường)
     1. Lý thuyết
     2. Thực hành
        + Bước 1: Khởi tạp project QT và openGL như ban đầu
        + Bước 2: Chỉnh sửa trong widget

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include <QWidget>  #include <QOpenGLShaderProgram>  #include <QTimer>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit **AXBOpenGLWidget**(QWidget \*parent = nullptr);  ~***AXBOpenGLWidget***();  enum **Shape**{None,Rect,Circle,Triangle};  void **drawShape**(Shape shape);  void **setWireframe**(bool wireframe);  protected:  void ***initializeGL***();  void ***resizeGL***(int w, int h);  void ***paintGL***();  public slots:  void **on\_timeout**();  private:  unsigned int VAO,VBO,EBO;  Shape m\_shape;  QOpenGLShaderProgram shaderProgram;  QTimer timer;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QDebug>  #include <QTime>  #include <QtMath>  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget( parent)  {  timer.start(100);  connect(&timer,SIGNAL(timeout()), this,SLOT(on\_timeout()));  }  void AXBOpenGLWidget::**on\_timeout**()  {  if (m\_shape==None) return;  makeCurrent();  int timeValue= QTime::currentTime().second();  float greenValue =(sin(timeValue/2.0)+0.5f);  shaderProgram.bind();  shaderProgram.setUniformValue("ourColor",0.0f, greenValue,0.0f);  update();  doneCurrent();  }  AXBOpenGLWidget::~***AXBOpenGLWidget***()  {  makeCurrent();  glDeleteBuffers(1,&VBO);  glDeleteVertexArrays(1,&VAO);  doneCurrent();  }  void AXBOpenGLWidget::**drawShape**(AXBOpenGLWidget::Shape shape)  {  makeCurrent();  m\_shape= shape ;  update();  doneCurrent();  }  void AXBOpenGLWidget::**setWireframe**(bool wireframe)  {  makeCurrent();  if(wireframe)  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);  else  glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);  update();  doneCurrent();  }  float vertices[]= {  0.5f, 0.5f,0.0f,  0.5f, -0.5f, 0.0f,  -0.5f, -0.5f, 0.0f,  -0.5f, 0.5f, 0.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,3\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO) sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  // bước 5\_2: EBO  glGenBuffers(1,&EBO);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof (indices), indices, GL\_STATIC\_DRAW);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex,":/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment,":/Shaders/obj.frag");  success=shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  // shaderProgram.bind();  // shaderProgram.setUniformValue("ourColor",1.0f,1.0f,0.0f); // set vec3  glBindVertexArray(VAO);  switch (m\_shape) {  case Rect:  glDrawElements(GL\_TRIANGLES,6,GL\_UNSIGNED\_INT,&indices);  break;  }  } |

Kết quả



1. VAO và VBO
   * 1. Lý thuyết
        + Tọa độ
          - Normalized Device coordinates(NDC) màn hình hiển thị window của opengl. Tọa độ dưới,trái là tọa dộ [-1,-1] và tọa độ phải góc trên là [1,1].
          - Tọa độ NDC chính là tọa độ khi hiển thị ra màn hình.Quy trình: Tọa độ không gian-> Tính toán ra NDC -> gán tọa độ NDC biến gl\_position.
        + [VBO]
          - Câu hỏi làm cách nào để đưa dữ liệu vào VertexShader?Hay Dữ liệu mảng các position đều nằm trên CPU làm sao chuyển dữ liệu từ CPU sang GPU được.
          - Trên GPU có một khu vực để nhận dữ liệu.Đối tượng Vertex Buffer Objec(VB0- GL\_ARRAY\_BUFFER) sẽ chuyển dữ liệu từ CPU sang khu vực nhận dữ liệu của GPU.
          - OpenGL cho phép liên kết nhiều bộ đệm cùng một lúc.
        + [VAO]
          - Khi truyền mảng dữ liệu từ bộ nhớ CPU sang GPU thông qua VBO. Làm sao để giải thích cho GPU hiểu được quy cách mảng dữ liệu này lưu cái gì? Như vậy cần có them một đối tượng mới là Vertex Array Object(VAO).
          - VAO chính là 1 mảng các object mỗi một item tương ứng với một thuộc tính: Pos, color, normal,…
          - Để set thuộc tính này vào vertex buffer thông qua đoạn code: layout (loaction = [0,1,2…]) in [type] [ten\_bien];
          - VAO không phải lưu dữ liệu thực tế. Nó chỉ là một định nghĩa về cấu trúc dữ liệu đỉnh lưu trữ của các đỉnh của chúng ta. Hay nói các khác nó giải thích dữ liệu này tương ứng với thuộc tính này.
          - Hình ảnh minh họa:



* + 1. Một số phương thức hay sử dụng đối với VAO và VBO
       - Phương thức

|  |  |  |
| --- | --- | --- |
| **STT** | **Phương thức** | **Mô tả** |
| 1 | glVertexAttribPointer | Giải thích cho GPU về phân tích dữ liệu đẩy vào Attrib pointer của VAO |
| 2 | glBufferData | Đẩy dữ liệu vào VBO(vào GPU) |

* + - * Giải thích chi tiết
    1. Thực hành
       - Bước 1: Khởi tạo project kết hợp opengl và QT
       - Bước 2: Chỉnh code

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include <QWidget>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  …  private:  unsigned int VAO,VBO;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget( parent)  {  }  float vertices[]= {  -0.5f, -0.5f,0.0f,  0.5f, -0.5f, 0.0f,  0.0f, 0.5f, 0.0f  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  // 0 ở đây chính là  trong VAO  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,3\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO)  sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  }  void AXBOpenGLWidget::***paintGL***()  {  …  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES,0,3);  } |

Kết quả



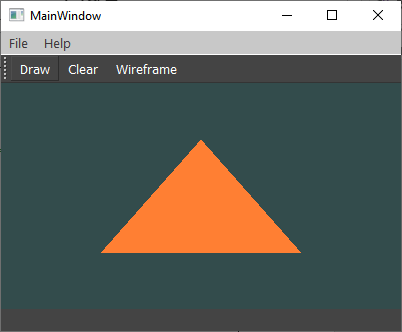
1. Shader
   1. Một số kiểu dữ liệu trong shader cơ bản

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Kiểu dữ liệu** | **Tiền tố** | **Vector** | **matrix** |
| 1 | float | f | vec3 |  |
| 2 | int | i |  |
| 3 | unsigned int | ui |  |
| 4 |  | 3f |  |
| 5 |  | fv |  |

* 1. Tạo và sử dụng shader cơ bản

|  |
| --- |
| class AXBOpenGLWidget : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  …  unsigned int VAO,VBO,shaderProgram;  }; |
| #include "axbopenglwidget.h"  #include <QDebug>  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget( parent)  {  }  const char \*vertexShaderSource = "#version 330 core\n"  "layout (location = 0) in vec3 aPos;\n"  "void main()\n"  "{\n"  "gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);\n"  "}\0";  const char \*fragmentShaderSource = "#version 330 core\n"  "out vec4 FragColor;\n"  "void main()\n"  "{\n"  "FragColor = vec4(1.0f, 0.5f, 0.2f,1.0f);\n"  "}\0";  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // …VAO và VBO  // ===Vertex Shader =====  unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);  glCompileShader(vertexShader);  int success; char infoLog[512];  glGetShaderiv(vertexShader,GL\_COMPILE\_STATUS, &success);  if(!success){  glGetShaderInfoLog(vertexShader,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog;  }  // == Fagment Shader ===  unsigned int fragShader = glCreateShader(GL\_FRAGMENT\_SHADER);  glShaderSource(fragShader, 1, &fragmentShaderSource, NULL);  glCompileShader(fragShader);  glGetShaderiv(fragShader,GL\_COMPILE\_STATUS, &success);  if(!success){  glGetShaderInfoLog(fragShader,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::FRAGMENT::COMPILATION\_FAILED\n" << infoLog;  }  // == Shader program ==  shaderProgram =glCreateProgram();  glAttachShader(shaderProgram, vertexShader);  glAttachShader(shaderProgram,fragShader );  glLinkProgram(shaderProgram);  glGetShaderiv(shaderProgram,GL\_LINK\_STATUS, &success);  if(!success){  glGetShaderInfoLog(shaderProgram,512,NULL ,infoLog);  qDebug() << "ERROR::SHADER::SHADERPROGRAM::COMPILATION\_FAILED\n" << infoLog;  }  // == Xóa Vertex shader, và fragment shader ==  glDeleteShader(vertexShader);  glDeleteShader(fragShader);  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  }  void AXBOpenGLWidget::***paintGL***()  {  …  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES,0,3);  } |

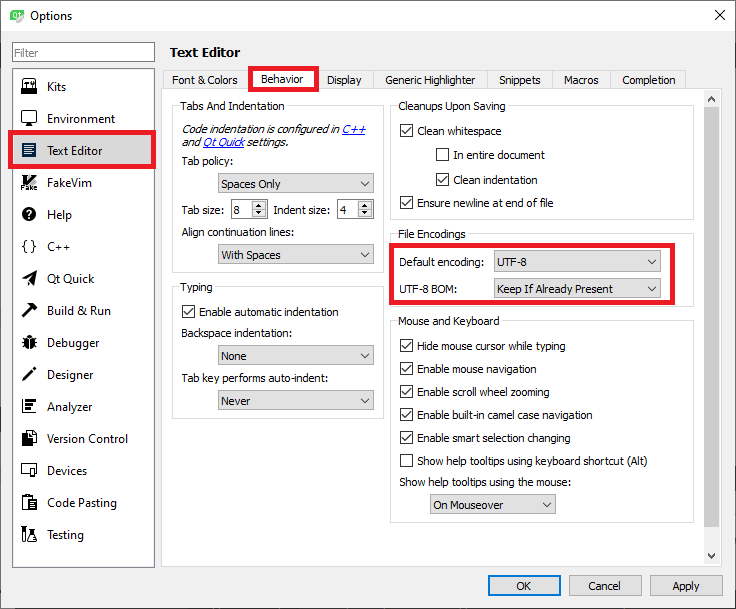
Kết quả



* 1. Tạo shader từ thư viện QT(QOpenglShaderProgram)
     1. Lý thuyết
     2. Một số phương thức của QopenGLShaderProgram cần chú ý

|  |  |  |
| --- | --- | --- |
| **STT** | **Phương thức** | **Mô tả** |
| 1 | addShaderFromSourceCode | Add mã code(string) vào |
| 2 | addShaderFromSourceFile | Add file shader bên ngoài |
| 3 | bink() |  |
| 4 | bind() |  |
| 5 | log() | Xem thông tin khi link shader có lỗi không |

Chú ý khi mà sử dụng phương thức fromSource file thì ta cần setup như sau. Nếu không setup thì khi chạy nó sẽ đọc code bị lỗi



* 1. Truyền dữ liệu từ code sang shader(uniform)

|  |  |  |
| --- | --- | --- |
| **Code chính** | **Shader** | **Mô tả** |
| shaderProgram.setUniformValue(  "ourColor",  1.0f,1.0f,0.0f); // set vec3 | uniform vec3 ourColor; | Truyền vector |
| shaderProgram.setUniformValue(  "offset",  1.0f); | uniform floar offset; | Truyền giá trị đơn |
| Qmaxtrix4x4 matrix;  //…  shaderProgram.setUniformValue(  "project", matrix); | Uniform mat4 project; | Truyền matran |
|  |  | Truyền phần tử thứ index của mảng |
|  |  | Truyền struct uniform |

* 1. Các phép toán đối với màu

|  |  |  |
| --- | --- | --- |
| **STT** | **Phép toán** | **Mô tả** |
| 1 | nhân | Màu nào nhân với màu trắng thì sẽ ra màu đó    Màu nào nhân với màu đen thì sẽ ra màu đen    Nhân màu với từng picture ta sẽ được picture tương ứng |
| 2 | Phép cộng | Màu nào cộng với màu đen thì sẽ ra được màu đấy |
| 3 | Nhân và cộng | ảnh tint mash là ảnh đen trắng |

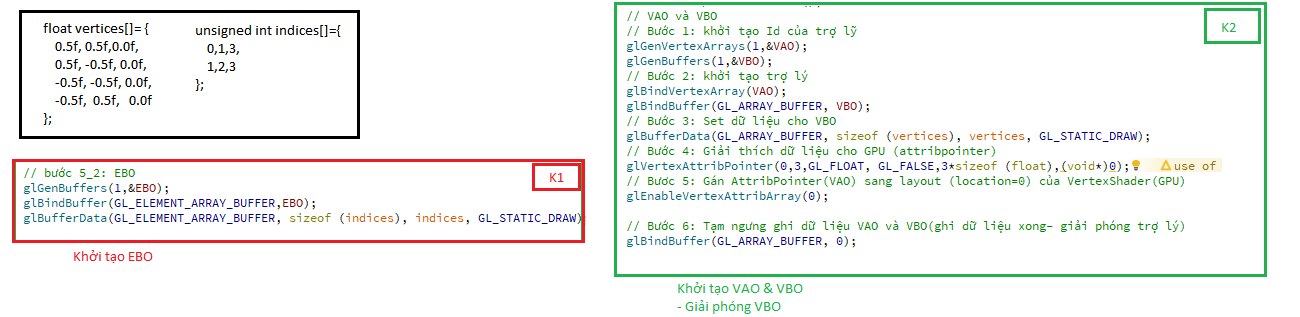
* 1. Thực hành
     1. Sử dụng text

|  |
| --- |
| #include <QOpenGLShaderProgram>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  …  QOpenGLShaderProgram shaderProgram;  }; |
| // Cách 1: từ code  const char \*vertexShaderSource = "#version 330 core\n"  "layout (location = 0) in vec3 aPos;\n"  "void main()\n"  "{\n"  "gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);\n"  "}\0";  const char \*fragmentShaderSource = "#version 330 core\n"  "out vec4 FragColor;\n"  "void main()\n"  "{\n"  "FragColor = vec4(1.0f, 0.5f, 0.2f,1.0f);\n"  "}\0";  bool success;  shaderProgram.addShaderFromSourceCode(QOpenGLShader::Vertex,vertexShaderSource);  shaderProgram.addShaderFromSourceCode(QOpenGLShader::Fragment,fragmentShaderSource);  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  // Cách 2: Sử dụng file trong resource  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex,”:/shader/test.vert”);  shaderProgram. addShaderFromSourceFile(QOpenGLShader::Fragment, ,”:/shader/test.frag”);  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log(); |
| // Sử dụng  shaderProgram.bind(); |

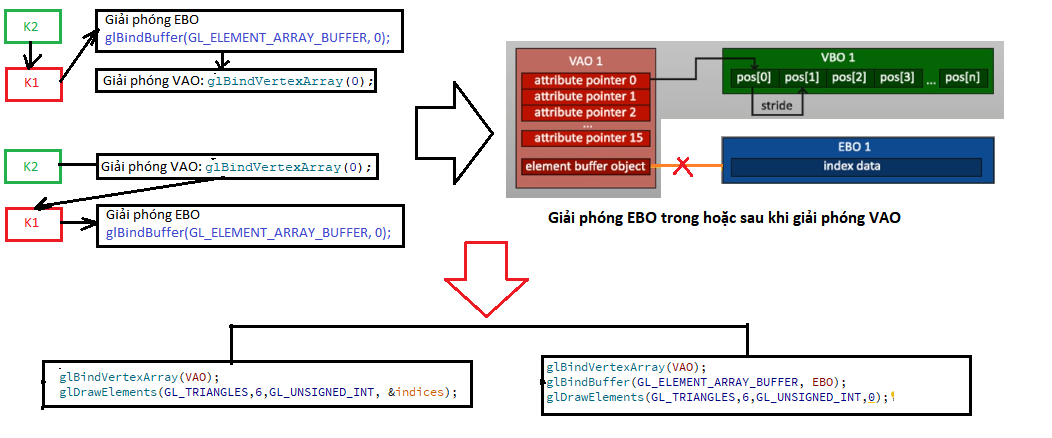
* + 1. Sử dụng file

|  |
| --- |
| void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  //…  bool success;  shaderProgram.addShaderFromSourceCode(QOpenGLShader::Vertex,":/shader/test.vert");  shaderProgram.addShaderFromSourceCode(QOpenGLShader::Fragment,":/shader/test.frag");  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  ..  shaderProgram.bind();  ..  } |

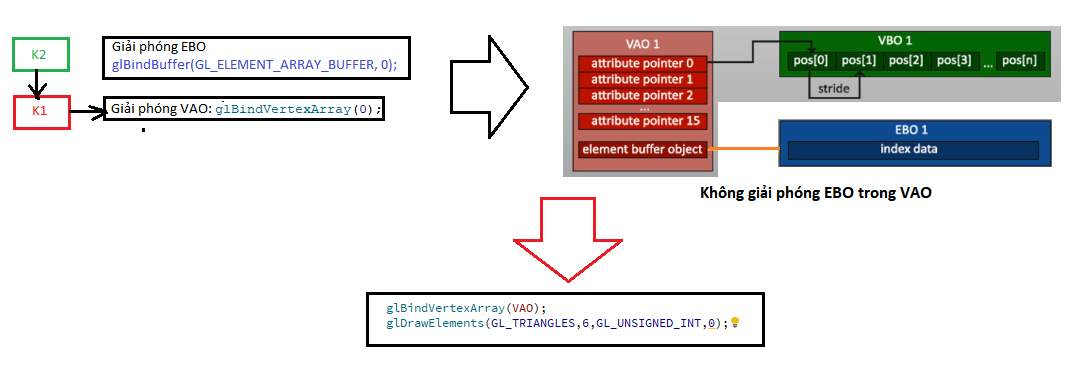
1. EBO
   1. Lý thuyết
      * + Element buffer object(EBO) hay index buffer object(IBO)
        + Khi vẽ các đối tượng thường vẽ bằng các pri… như vậy sẽ có nhiều đỉnh sẽ lặp lại. Ví dụ như vẽ 1 hcn được cấu tạo bởi 2 hình tam giác. Khi truyền vào VBO theo cách thông thường thì ta cần truyền vào 6 đỉnh. Cho dù hcn chỉ cần 4 đỉnh là đủ. Như vậy việc dư đỉnh là 50%. Nên họ mới sử dụng đối tượng EBO này.
        + Chỉ cần tryền vào VBO các đỉnh không trùng nhau sau đó dùng EBO này để quy định nối các điểm liên kết với nhau.
        + Khi vẽ đối tượng thay vì sử dụng glDrawArrays ta bắt buộc phải sử dụng phương thức sau: glDrawElements
   2. Chú ý khi sử dụng EBO có các trường hợp như sau khi vẽ đối tượng:
      1. Quy ước



* + 1. TH1: Giải phóng EBO trong hoặc sau khi giải phóng VAO.



* + 1. TH2: Không giải phong EBO, và EBO nằm trong VAO.



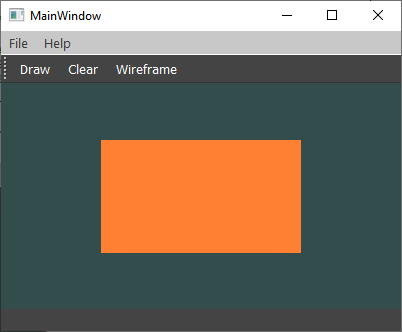
* 1. Thực hành
     1. Sửa dụng VAO và VBO vẽ hình chữ nhật

|  |
| --- |
| float vertices[]= {  // first triangle  0.5f, 0.5f,0.0f,  0.5f, -0.5f, 0.0f,  -0.5f, 0.5f, 0.0f,  //second triangle  0.5f, -0.5f, 0.0f,  -0.5f, -0.5f, 0.0f,  -0.5f, 0.5f, 0.0f  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,3\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO) sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  // === Shader =====  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES,0,6);  } |

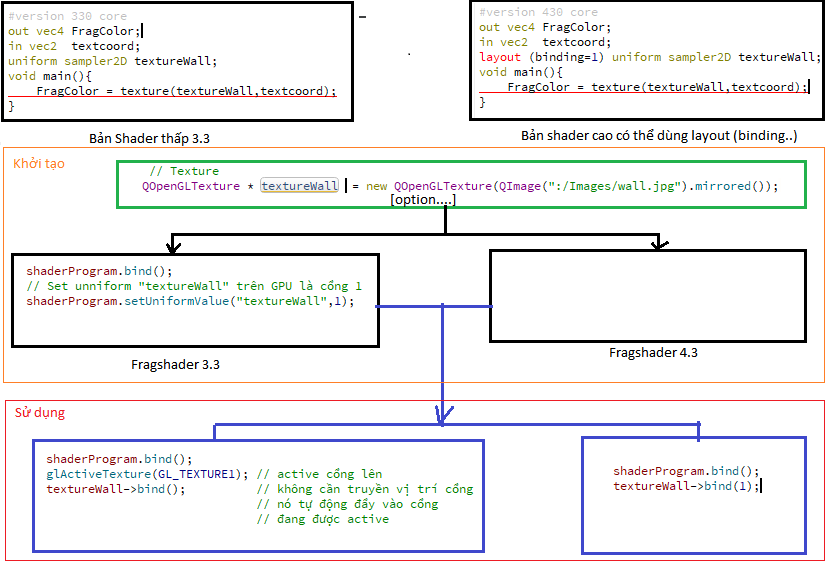
* + 1. Sử dụng EBO vẽ hình chữ nhật

|  |
| --- |
| float vertices[]= {  0.5f, 0.5f,0.0f,  0.5f, -0.5f, 0.0f,  -0.5f, -0.5f, 0.0f,  -0.5f, 0.5f, 0.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,3\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO) sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);    // bước 5\_2: EBO  glGenBuffers(1,&EBO);  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof (indices), indices, GL\_STATIC\_DRAW);  );  // EBO chưa giải phòng và EBO nằm trong VAO  glBindVertexArray(0);  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  glDrawElements(GL\_TRIANGLES,6,GL\_UNSIGNED\_INT,0);  //Chưa hủy bỏ liên keert EBO trong VAO nên được sử dụng đoạn code này  } |

Kết quả



1. Texture
   1. Load texture
      1. Lý thuyết
         * Cách khởi tạo, load texture và đẩy vào GPU (shader)



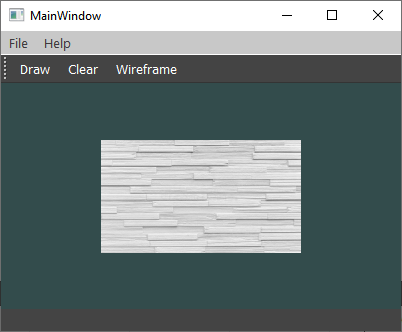
* + - * Số cổng của texture của GPU từ GL\_TEXTURE0 đến GL\_TEXTURE15. Ví dụ TEXTURE5= TEXTRE0+5
      * Một điểm ảnh trong texture được gọi là texel(khác pixel).
      * Phương S là phương ngang còn phương T là phương dọc.
      * Một số function xử lý liên quan đến ảnh trong fragmentshader

|  |  |  |
| --- | --- | --- |
| **STT** | **Phương thức** | **Mô tả** |
| 1 | uniform sampler2D textureWall;  in vec2 textcoord;  FragColor = texture(textureWall,textcoord); | Gán ảnh vào textcoord |
| 2 | FragColor = texture(textureWall,textcoord)\* vec4(1.0f,0.0f,0.0f,1.0f); | Mix texture với color |
| 3 | FragColor = mix(  texture(textureWall,textcoord),  texture(textureWall2,textcoord),  0.5  ); | Mix 2 texture với nhau |

* + 1. Thực hành

|  |
| --- |
| class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  …  private:  ..  QOpenGLTexture \* textureWall;  }; |
| #include "axbopenglwidget.h"  #include <QDebug>  #include <QTime>  #include <QtMath>  #include <QImage>  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget( parent)  {  ..  }  float vertices[]= {  // Position // Text coord  0.5f , 0.5f, 0.0f, 1.0f,1.0f,  0.5f ,-0.5f, 0.0f, 1.0f,0.0f,  -0.5f,-0.5f, 0.0f, 0.0f,0.0f,  -0.5f, 0.5f, 0.0f, 0.0f, 1.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // VAO và VBO  ..  // Texture  textureWall = new QOpenGLTexture(QImage(":/Images/wall.jpg").mirrored());  shaderProgram.bind();  // Set unniform "textureWall" trên GPU là cổng 1  shaderProgram.setUniformValue("textureWall",1);  ..  }  void AXBOpenGLWidget::***paintGL***()  {  ..  shaderProgram.bind();  // Đẩy dữ liệu image vào cổng 1  textureWall->bind(1);  glBindVertexArray(VAO);  switch (m\_shape) {  case Rect:  glDrawElements(GL\_TRIANGLES,6,GL\_UNSIGNED\_INT,&indices);  break;  }} |
| // vert shader  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec2 atexCrd;  out vec2 textcoord;  void main(){  gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);  textcoord= atexCrd;  } |
| // frag shader  #version 330 core  out vec4 FragColor;  in vec2 textcoord;  uniform sampler2D textureWall;  void main(){  FragColor = texture(textureWall,textcoord);  } |

Kết quả



* 1. Option Texture:Wrap – Lặp lại của texture - GL\_TEXTURE\_WRAP\_[S|T]
     1. Lý thuyết
        + Có 4 tham số để wrap texture

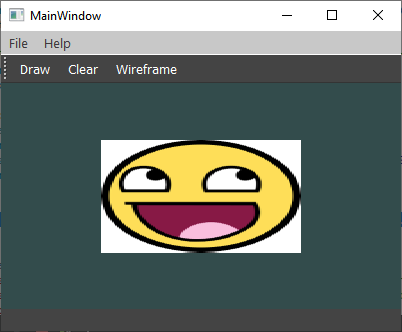
|  |  |  |
| --- | --- | --- |
| **STT** | **Thông số** | **Mô tả** |
| 1 | GL\_REPEAT | Nó sẽ lặp đi lặp lại cho đến hết pattern. Nếu ảnh nó nhỏ thì tự động nó thêm ảnh vào đến hết pattern mới thôi. |
| 2 | GL\_MIRRORED\_REPEAT | Giống như repeat có điều ảnh nó sẽ mirror theo cạnh đó |
| 3 | GL\_CLAMP\_TO\_EDGE | Tọa độ nhỏ hơn 0 và lớn hơn 1 được thiết lập đến 0 và 1 tương ứng. |
| 4 | GL\_CLAMP\_TO\_BORDER | Nội dung bên ngoài [0..1] sẽ được chỉ định một số màu đường viền được chỉ định. |



* + - * Chú ý khi set cho texture thì mình phải textture->bind(index) trước khi set thuộc tính
    1. Thực hành
       - Normal

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.5f , 0.5f, 0.0f, 1.0f,1.0f,  0.5f ,-0.5f, 0.0f, 1.0f,0.0f,  -0.5f,-0.5f, 0.0f, 0.0f,0.0f,  -0.5f, 0.5f, 0.0f, 0.0f, 1.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  // Texture  textureWall = new QOpenGLTexture(QImage(":/Images/awesomeface.png").mirrored()); |

Kết quả



* + - * GL\_REAPEAT

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.5f , 0.5f, 0.0f, 2.0f,2.0f,  0.5f ,-0.5f, 0.0f, 2.0f,-1.0f,  -0.5f,-0.5f, 0.0f, -1.0f,-1.0f,  -0.5f, 0.5f, 0.0f, -1.0f, 2.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  // Texture  textureWall = new QOpenGLTexture(QImage(":/Images/awesomeface.png").mirrored()); |
| shaderProgram.bind();  glActiveTexture(GL\_TEXTURE1); // active cổng lên  textureWall->bind(); // không cần truyền vị trí cổng  // nó tự động đẩy vào cổng  // đang được active  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT); |

Kết quả



* + - * GL\_MIRROR\_REPEAT

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.5f , 0.5f, 0.0f, 2.0f,2.0f,  0.5f ,-0.5f, 0.0f, 2.0f,-1.0f,  -0.5f,-0.5f, 0.0f, -1.0f,-1.0f,  -0.5f, 0.5f, 0.0f, -1.0f, 2.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  // Texture  textureWall = new QOpenGLTexture(QImage(":/Images/awesomeface.png").mirrored()); |
| shaderProgram.bind();  glActiveTexture(GL\_TEXTURE1); // active cổng lên  textureWall->bind(); // không cần truyền vị trí cổng  // nó tự động đẩy vào cổng  // đang được active  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_MIRRORED\_REPEAT);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_MIRRORED\_REPEAT); |

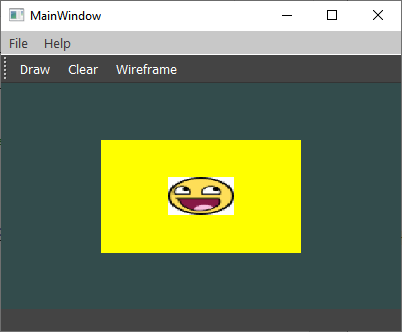
Kết quả



* + - * GL\_CLAM\_TO\_EDGE
      * GL\_CLAMP\_TO\_BORDER

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.5f , 0.5f, 0.0f, 2.0f,2.0f,  0.5f ,-0.5f, 0.0f, 2.0f,-1.0f,  -0.5f,-0.5f, 0.0f, -1.0f,-1.0f,  -0.5f, 0.5f, 0.0f, -1.0f, 2.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  };  // Texture  textureWall = new QOpenGLTexture(QImage(":/Images/awesomeface.png").mirrored()); |
| shaderProgram.bind();  glActiveTexture(GL\_TEXTURE1); // active cổng lên  textureWall->bind(); // không cần truyền vị trí cổng  // nó tự động đẩy vào cổng  // đang được active  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_BORDER);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_BORDER);  // Set color to border  float borderColor[]={1.0f,1.0f,0.0,1.0f};  glTexParameterfv(GL\_TEXTURE\_2D,GL\_TEXTURE\_BORDER\_COLOR, borderColor ); |

Kết quả



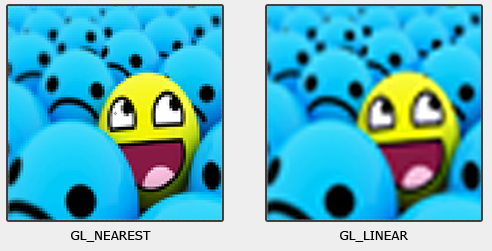
* 1. Option Texture: Filter – Độ phân giải của texture – GL\_TEXTURE\_[MIN|MAG]\_FILTER
     1. Lý thuyết
        + Các chế độ thiết lập

|  |  |  |
| --- | --- | --- |
| **STT** | **Thông số** | **Mô tả** |
| 1 | GL\_NEAREST | Lấy màu của pixel gần nhất trên ảnh của texture.[Nhìn rõ các ô màu trên ảnh] |
| 2 | GL\_LINEAR | Nó sẽ lấy 4 pixels liền kề xung quanh tọa độ UV và tính toán giá trị trung bình để cho ra màu sắt.  Công thức: 50%a + 20%b + 20%c +10%d |

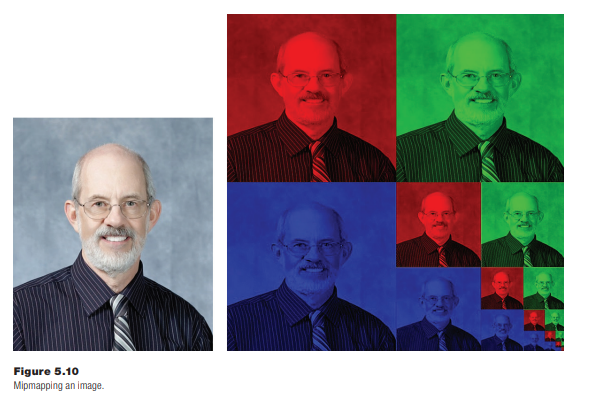
* + - * MAG phóng đại ảnh còn MIN là thu nhỏ ảnh
    1. Thực hành

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.9f , 0.9f, 0.0f, 1.0f,1.0f,  0.9f ,-0.9f, 0.0f, 1.0f,0.0f,  -0.9f,-0.9f, 0.0f, 0.0f,0.0f,  -0.9f, 0.9f, 0.0f, 0.0f, 1.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  }; |
| shaderProgram.bind();  glActiveTexture(GL\_TEXTURE1); // active cổng lên  textureWall->bind(); // không cần truyền vị trí cổng  // nó tự động đẩy vào cổng  // đang được active  // Option texture wrap nếu có  glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MIN\_FILTER,GL\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MAG\_FILTER,GL\_LINEAR); |

Kết quả



* 1. Option Texture: Mipmap – Trong texture có các đối tượng giống nhau nhưng khác nhau về kích thước
     1. Lý thuyết
        + Mipmapping hoạt động theo một cơ chế thông minh để lưu trữ một loạt các các bản sao có độ phân giải thấp hơn của cùng một hình. Điều này đạt được bằng cách lưu trữ các thành phần R, G và B của hình ảnh riêng biệt trong ba phần tư không gian texture image, sau đó lặp lại xử lý trong một phần tư không gian hình ảnh còn lại cho cùng một hình ảnh tại một phần tư độ phân giải ban đầu. Việc chia nhỏ này lặp lại cho đến khi phần còn lại góc phần tư quá nhỏ để chứa bất kỳ dữ liệu hình ảnh hữu ích nào.



* + - * Thông thường khi sử dụng mipmap thì luôn sử dụng đoạn code cố định sau

|  |
| --- |
| glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MIN\_FILTER,GL\_LINEAR\_MIPMAP\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MAG\_FILTER,GL\_LINEAR); |

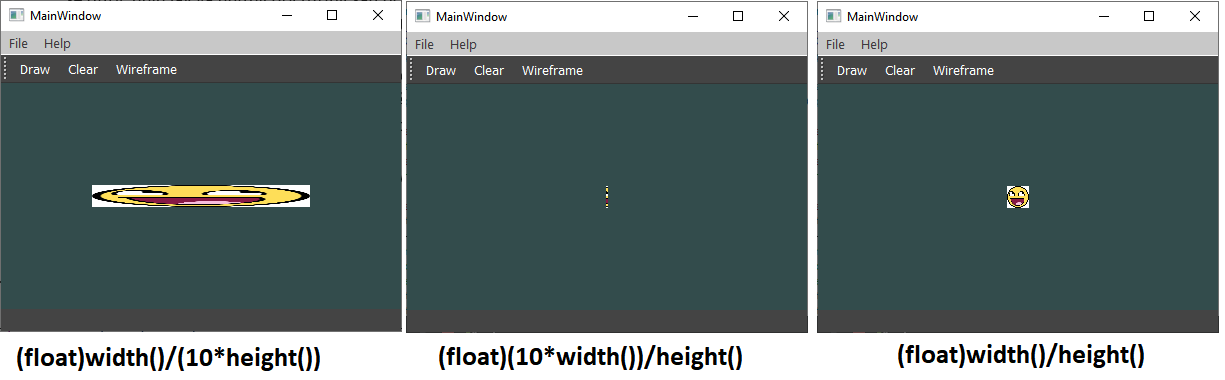
* + - * Trước khi texturte->bind(index) thì cần sử dụng phương thức: texture->
      * Các thông số

|  |  |  |
| --- | --- | --- |
| **STT** | **Thông số** | **Mô tả** |
| 1 | GL\_NEAREST\_MIPMAP\_NEAREST | Chọn mipmap có độ phân giải tương tự nhất của khu vực trong số các pixel được tạo texture. Sau đó, nó nhận được texel gần nhất với mong muốn texture coordinate. |
| 2 | GL\_LINEAR\_MIPMAP\_NEAREST | Chọn mipmap có độ phân giải tương tự nhất của khu vực trong số các pixel được tạo texture. Sau đó, nó nội suy bốn texel gần nhất với texture coordinate. Đây được gọi là “linear filtering - lọc tuyến tính”. |
| 3 | GL\_NEAREST\_MIPMAP\_LINEAR | Chọn hai mipmap có độ phân giải gần nhất với độ phân giải của khu vực  trong số các pixel được tạo texture. Sau đó, nó nhận được texel gần nhất với texture coordinates từ mỗi mipmap và nội suy chúng. Cái này được gọi là " bilinear filtering - Lọc song tuyến." |
| 4 | GL\_LINEAR\_MIPMAP\_LINEAR | Chọn hai mipmap có độ phân giải gần nhất với độ phân giải của khu vực  trong số các pixel được tạo texture. Sau đó, nó nội suy bốn texel gần nhất  trong mỗi mipmap và nội suy hai kết quả đó. Cái này được gọi là  ““trilinear filtering - Lọc ba tuyến” . “Trilinear filtering - Lọc ba tuyến” thường thích hợp hơn, vì mức độ hòa trộn thấp hơn thường tạo ra các tạo tác, chẳng hạn như sự phân cách có thể nhìn thấy giữa các mức mipmap. |

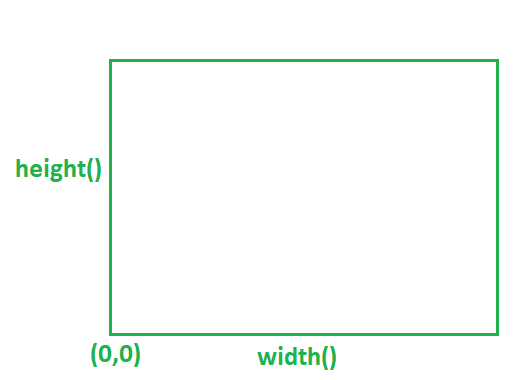
* + 1. Thực hành

|  |
| --- |
| float vertices[]= {  // Position // Text coord  0.9f , 0.9f, 0.0f, 1.0f,1.0f,  0.9f ,-0.9f, 0.0f, 1.0f,0.0f,  -0.9f,-0.9f, 0.0f, 0.0f,0.0f,  -0.9f, 0.9f, 0.0f, 0.0f, 1.0f  };  unsigned int indices[]={  0,1,3,  1,2,3  }; |
| shaderProgram.bind();  glActiveTexture(GL\_TEXTURE1); // active cổng lên  textureWall->generateMipMaps(); // Kích hoạt cho phép gen mipmap  textureWall->bind(); // không cần truyền vị trí cổng  // nó tự động đẩy vào cổng  // đang được active  // Option texture wrap nếu có  glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MIN\_FILTER,GL\_LINEAR\_MIPMAP\_LINEAR);  glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MAG\_FILTER,GL\_NEAREST); |

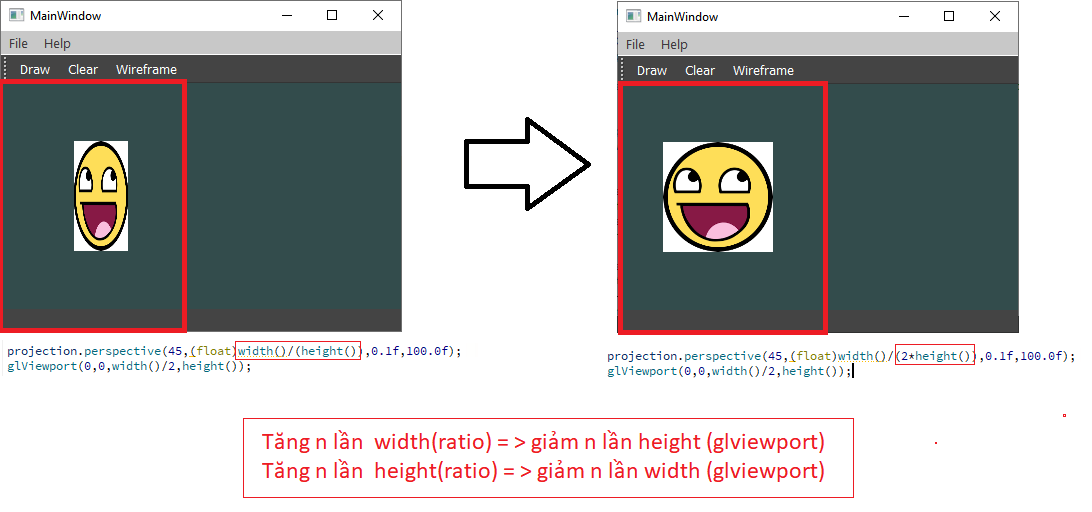
1. Ratio và glviewport
   1. Ratio
      * + Tỷ lệ khung hình chuẩn nhấy khi sử dụng phép chiếu perspective là width/height.Tức là khi vẽ hình vuông trong không gian thì nó sẽ hiện ra được hình vuông
          - width/(n\*height): n lần của height= 1 đơn vị của width (hình bị bẹp – bề rộng to hơn chiều cao).
          - n\*width/height: phải cần n đơn vị của width mới bằng 1 đơn vị của height(hình sẽ bị gầy đi).



* 1. glViewport
     + - Thay vì vẽ toàn bộ dữ liệu trên màn hình thì ta sử dụng viewport để vẽ toàn bộ dữ liệu trên 1 phần của màn hình.
         * Thông thường màn hình hiển thị toàn bộ dữ liệu mặc định là glviewport(0,0,width,height)



* + - * + Bây giờ người dùng muốn hiển thị dữ liệu ở 1 nửa màn hình=> width= 0.5\*width() thì ta sử dụng lệnh glviewport(0,0,width()/2,height(). Nhưng khi vẽ đối tượng ra thì bề rộng của đối tượng gấp 2 lần so với bình thường nên thường khi set glviewport không theo mặc định thì ta cần phải set thêm ratio trong phép chiếu.



1. Toạ độ
   * + - Mclip= Mproject\*Mview\*Mmodel\*Pos.Kết quả sẽ là 1 tọa độ mới trong không gian clip. Trong đó w được giới hạn bởi đoạn [-1,1] ngoài phạm vi này thì sẽ không được hiển thị trong không gian clip. Tọa độ w này càng nhỏ chứng tỏ vật thể càng gần camera và ngược lại.
       - Khi vẽ các đối tượng 3D. thì sẽ có những pixel bị trùng,khi hiển thị nó sẽ hiển thị ra màu cuối cùng (gi đè màu trên cùng pixel). Để đúng với thực tế những đối tượng nằm trước sẽ được nhìn lấy và những đối tượng sau bị che khuất sẽ không được nhìn thấy thì ta phải sử dụng depth buffer. Khi sử dụng Depth buffer thì nó sẽ giữ lại pixel nhìn thấy đầu tiên.
2. Camera
   1. Lý thuyết
   2. Thực hành
      1. Tạo class camera

|  |
| --- |
| #ifndef CAMERA\_H  #define CAMERA\_H  #include <QVector3D>  #include <QMatrix4x4>  enum **Camera\_Movement** {  FORWARD,  BACKWARD,  LEFT,  RIGHT  };  // Default camera values  const float YAW = -90.0f;  const float PITCH = 0.0f;  const float SPEED = 2.5f;  const float SENSITIVITY = 0.1f;  const float ZOOM = 45.0f;  class **Camera**  {  public:  // camera Attributes  QVector3D Position;  QVector3D Front;  QVector3D Up;  QVector3D Right;  QVector3D WorldUp;  // euler Angles  float Yaw;  float Pitch;  // camera options  float MovementSpeed;  float MouseSensitivity;  float Zoom;  // constructor with vectors  **Camera**(QVector3D position =QVector3D(0.0f, 0.0f, 0.0f), QVector3D up = QVector3D(0.0f, 1.0f, 0.0f), float yaw = YAW, float pitch = PITCH);  // constructor with scalar values  **Camera**(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch);  QMatrix4x4 **GetViewMatrix**();  // processes input received from any keyboard-like input system. Accepts input parameter in the form of camera defined ENUM (to abstract it from windowing systems)  void **ProcessKeyboard**(Camera\_Movement direction, float deltaTime);  // processes input received from a mouse input system. Expects the offset value in both the x and y direction.  void **ProcessMouseMovement**(float xoffset, float yoffset, bool constrainPitch = true);  // processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis  void **ProcessMouseScroll**(float yoffset);  private:  // calculates the front vector from the Camera's (updated) Euler Angles  void **updateCameraVectors**();  float **DegToRadian**(float degree);  };  #endif // CAMERA\_H |
| #include "camera.h"  Camera::**Camera**(QVector3D position, QVector3D up , float yaw , float pitch )  : Front(QVector3D(0.0f, 0.0f, -1.0f)),  MovementSpeed(SPEED),  MouseSensitivity(SENSITIVITY),  Zoom(ZOOM)  {  Position = position;  WorldUp = up;  Yaw = yaw;  Pitch = pitch;  updateCameraVectors();  }  // constructor with scalar values  Camera::**Camera**(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch)  : Front(QVector3D(0.0f, 0.0f, -1.0f)),  MovementSpeed(SPEED),  MouseSensitivity(SENSITIVITY),  Zoom(ZOOM)  {  Position = QVector3D(posX, posY, posZ);  WorldUp = QVector3D(upX, upY, upZ);  Yaw = yaw;  Pitch = pitch;  updateCameraVectors();  }  QMatrix4x4 Camera::Camera::**GetViewMatrix**()  {  QMatrix4x4 mat;  mat.setToIdentity();  mat.lookAt(Position, Position + Front, Up);  return mat;  }  void Camera::**ProcessKeyboard**(Camera\_Movement direction, float deltaTime)  {  float velocity = MovementSpeed \* deltaTime;  if (direction == FORWARD)  Position += Front \* velocity;  if (direction == BACKWARD)  Position -= Front \* velocity;  if (direction == LEFT)  Position -= Right \* velocity;  if (direction == RIGHT)  Position += Right \* velocity;  }  void Camera::**ProcessMouseMovement**(float xoffset, float yoffset, bool constrainPitch)  {  xoffset \*= MouseSensitivity;  yoffset \*= MouseSensitivity;    Yaw += xoffset;  Pitch += yoffset;    // make sure that when pitch is out of bounds, screen doesn't get flipped  if (constrainPitch)  {  if (Pitch > 89.0f)  Pitch = 89.0f;  if (Pitch < -89.0f)  Pitch = -89.0f;  }  // update Front, Right and Up Vectors using the updated Euler angles  updateCameraVectors();  }  void Camera::**ProcessMouseScroll**(float yoffset)  {  Zoom -= (float)yoffset;  if (Zoom < 1.0f)  Zoom = 1.0f;  if (Zoom > 45.0f)  Zoom = 45.0f;  }  const float PI=3.141589;  void Camera::**updateCameraVectors**()  {  // calculate the new Front vector  QVector3D front;    front.setX( cos(DegToRadian(Yaw)) \* cos(DegToRadian(Pitch)));  front.setY(sin(DegToRadian(Pitch)));  front.setZ( sin(DegToRadian(Yaw)) \* cos(DegToRadian(Pitch)));  front.normalize();  Front = front;  // also re-calculate the Right and Up vector  Right =QVector3D::crossProduct(Front, WorldUp);  Right.normalize();  // normalize the vectors, because their length gets closer to 0 the more you look up or down which results in slower movement.  Up = QVector3D::crossProduct(Right, Front);  Up.normalize();  }  float Camera::**DegToRadian**(float degree)  {  return degree\*PI/180;  } |

* + 1. Áp dụng class camera vào openglWidget

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include "camera.h"  #include <QPointF>  #include <QOpenGLShaderProgram>  #include <QTimer>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit **AXBOpenGLWidget**(QWidget \*parent = nullptr);  private slots:  void **on\_TimeOut**();  protected:  void ***initializeGL***();  void ***resizeGL***(int w, int h);  void ***paintGL***();  // Event  void ***mousePressEvent***(QMouseEvent \*event);  void ***mouseMoveEvent***(QMouseEvent \*event);  void ***wheelEvent***(QWheelEvent \*event);  void ***keyPressEvent***(QKeyEvent \*event);  private:  unsigned int VAO, VBO;  Camera m\_camera;  QOpenGLShaderProgram shaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QMouseEvent>  #include <QWheelEvent>  #include <QKeyEvent>  #include <QDebug>  QPoint lastPos(0,0);  int perFrame = 100;// delta time / frame  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget(parent)  {  /\*  Cho phép kích hoạt sử dụng event keypress  nếu khong có dòng này mà vẫn có code  event key press thì nó cũng không chạy được  \*/  setFocusPolicy(Qt::StrongFocus);  /\* Cho phép kích hoạt sự kiện liên quan đến mouse\*/  //setMouseTracking(true);  m\_camera= Camera(QVector3D(0.0f, 0.0f, 3.0f));  m\_timer.setInterval(perFrame);  connect(&m\_timer, SIGNAL(timeout()), this, SLOT(on\_TimeOut()));  m\_timer.start();  }  void AXBOpenGLWidget::**on\_TimeOut**()  {  update();  }  float vertices[] = {  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  0.5f, -0.5f, -0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  // VAO và VBO  // Bước 1: khởi tạo Id của trợ lỹ  glGenVertexArrays(1,&VAO);  glGenBuffers(1,&VBO);  // Bước 2: khởi tạo trợ lý  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  // Bước 3: Set dữ liệu cho VBO  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);  // Bước 4: Giải thích dữ liệu cho GPU (attribpointer)  glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,5\*sizeof (float),(void\*)0);  // Bươc 5: Gán AttribPointer(VAO) sang layout (location=0) của VertexShader(GPU)  glEnableVertexAttribArray(0);  // Bước 6: Tạm ngưng ghi dữ liệu VAO và VBO(ghi dữ liệu xong- giải phóng trợ lý)  glBindBuffer(GL\_ARRAY\_BUFFER, 0);  glBindVertexArray(0);  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  Q\_UNUSED(w)  Q\_UNUSED(h)  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  }  void AXBOpenGLWidget::***mousePressEvent***(QMouseEvent \*event)  {  lastPos = event->pos();  }  void AXBOpenGLWidget::***mouseMoveEvent***(QMouseEvent \*event)  {  auto posIn= event->pos();  // reversed since y-coordinates go from bottom to top  int xoffset = posIn.x() - lastPos.x();  int yoffset = lastPos.y() - posIn.y();  lastPos=posIn;  m\_camera.ProcessMouseMovement(xoffset, yoffset);  update();  }  void AXBOpenGLWidget::***wheelEvent***(QWheelEvent \*event)  {  m\_camera.ProcessMouseScroll( event->angleDelta().y()/120);  update();  }  void AXBOpenGLWidget::***keyPressEvent***(QKeyEvent \*event)  {  float detatime2= 2.5\*perFrame/1000;  switch (event->key()) {  case Qt::Key\_W:m\_camera.ProcessKeyboard(FORWARD,detatime2); break;  case Qt::Key\_S:m\_camera.ProcessKeyboard(BACKWARD,detatime2); break;  case Qt::Key\_D:m\_camera.ProcessKeyboard(RIGHT,detatime2); break;  case Qt::Key\_A:m\_camera.ProcessKeyboard(LEFT,detatime2); break;  }  update();  } |
| // obj.vert  #version 330 core  layout (location = 0) in vec3 aPos;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  void main(){  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| // obj.frag  #version 330 core  out vec4 FragColor;  void main(){  FragColor = vec4(1.0f, 0.5f, 0.2f,1.0f);  }; |

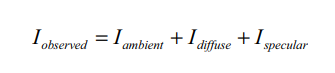
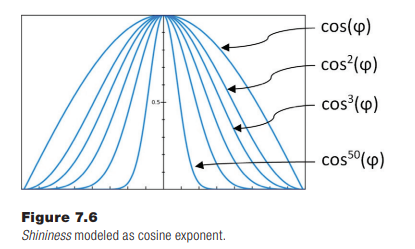
Kết quả

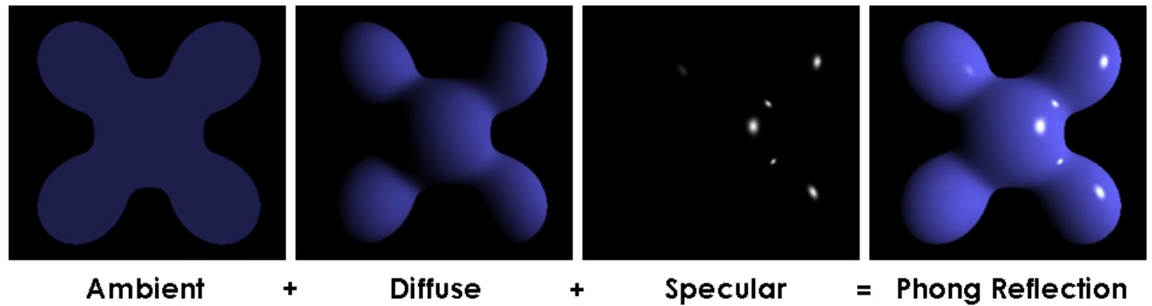


1. Light
   1. Lưu ý
      * + Khi truyền vector normal của vertex vào shader ta cần chú ý:
          - Khi mà kích cỡ của cửa sổ window không bị xê dịch thì vector normal tính toán chính bằng vector normal truyền vào shader.
          - Khi kích cỡ cửa sổ window bị co dãn thì vector normal khi tính toán sẽ không phải vector normal truyền vào mà cần tính lại theo công thức sau:

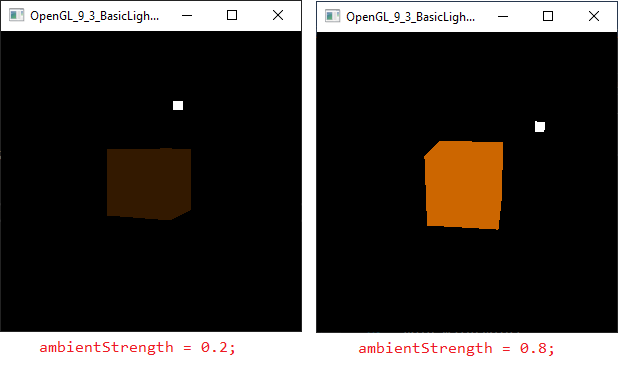
|  |
| --- |
| Normal = mat3(transpose(inverse(model))) \* aNormal; |

* + - * + Link tính toán lại vector normal: <http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/>
  1. Mô hình chiếu sáng ASD
     1. Hình ảnh





* + 1. Ambient(A)
       - Ánh sáng phát ra từ chính bản thân vật thể
       - Một vật thể trong không gian nó chịu ảnh hưởng bởi cường độ ánh sáng.
       - Ví dụ: 1 cái áo trắng khi trời nắng(ambientStrength=0.8) thì cái áo đó rất là sáng. Nhưng khi chiều về tối (ambientStrength=0.2) thì trông nó rất là tối.
       - Thông số này chịu ảnh hưởng: Màu ánh sáng(lightColor); Cường độ sáng(ambientStrength); và màu của vật thể(objectColor)
       - Công thức: FragColor(fragmentShader) = ambientStrength\*lightColor\*objectColor



* + - * Ví dụ:

|  |
| --- |
| //file modelShader.frag  #version 430  out vec4 FragColor;  in vec2 textCoord;  in vec3 normal;  uniform vec3 lightColor;  uniform vec3 objectColor;  void main(void)  {  // Cường độ sáng  float ambientStrength = 0.8;  vec3 result= lightColor \* ambientStrength \* objectColor;  FragColor = vec4(result, 1.0f);  } |

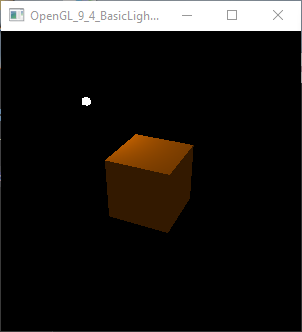
* + 1. Diffuse(D)
       - Phản xạ,khuếch tán dựa vào góc tới của ánh sáng(ánh sánh chiếu đến đâu thì sáng đến đó)
       - Định luật lambert’s cosin law

|  |  |
| --- | --- |
|  | Cường độ phải lớn hơn 0 tức cos>0 => -90<phi<90 =>    Diffuse component = max(dot(N,L),0)\*lightColor |

* + - * Code

|  |  |
| --- | --- |
| // file modelShader.vert  #version 430  layout (location = 0) in vec3 aPos;  layout (location=1) in vec3 aColor;  layout (location=2) in vec2 aTextCoord;  layout (location=3) in vec3 aNormal;  uniform mat4 projection;  uniform mat4 model;  uniform mat4 view;  out vec2 textCoord;  out vec3 normal;  out vec3 modelPos;  void main()  {  textCoord= aTextCoord;  normal= aNormal;  modelPos = (model \* vec4(aPos,1.0f)).xyz ;  gl\_Position= projection \*view\*model\* vec4(aPos,1.0f);  } | // file modelShader.frag  #version 430  out vec4 FragColor;  in vec2 textCoord;  in vec3 normal;  in vec3 modelPos;  uniform vec3 lightColor;  uniform vec3 objectColor;  uniform vec3 lightPos;  void main(void)  {  // Cường độ sáng  float ambientStrength = 0.2;  vec3 ambient= lightColor \* ambientStrength ;  // diffuse  **vec3 norm = normalize(normal);**  **vec3 lightDir = normalize(lightPos-modelPos);**  **float diff = max(dot(norm,lightDir),0.0f);**  **vec3 diffuse = lightColor \* diff;**  // combine  vec3 result = (ambient + diffuse)\* objectColor;  FragColor = vec4(result, 1.0f);  } |

Kết quả



* + 1. Specular(S) độ bóng của vật thể
       - Thông số

|  |  |  |
| --- | --- | --- |
| **STT** | **Thông số** | **Mô tả** |
| 1 | specularStrength | specularStrength càng cao thì càng nhìn rõ hình dạng ánh sáng trên vật thể. |
| 2 | Shininess | Mức độ loang của ánh sáng khi chiếu lên bề mặt vật thể. Thường được mô phỏng với một hàm falloff. Biểu đồ sau mô tả mối quan hệ giữa góc phi và shininess. Số mũ càng cao suy giảm càng mạnh dẫn đến phản xạ của ánh sáng càng nhỏ. Số mũ n được gọi là shininess. |
| 3 | Emissiveness | Vật liệu phát ra ánh sáng của chính nó(vật liệu phát quang) |

* + - * Hai loại mô hình chiếu specular
        + Gouraud shading(henri gouraud -1971)

|  |  |
| --- | --- |
|  | R: normalized vector of light reflected off the surface  C: Normalized vector from point to camera  K: constant exponent(vd: 32,64,128); larger k make smaller hightlight  Specular component= exp(max(dot(R,c),0),k)\*lightColor. |

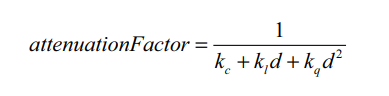
* + - * + Blin-phong specluar light

|  |  |
| --- | --- |
|  | N: normal at point of surface  H: normalized ‘half’ (bisecting) vector between C (camera) and L (light)  K: constant exponent(vd: 32,64,128); larger k make smaller hightlight  Specular component = exp(max(dot(N,H),0),k)\*lightColor |

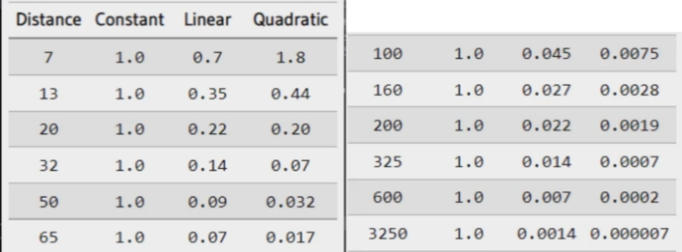
* + - * + Mô hình

|  |  |
| --- | --- |
|  | R đại diện cho hướng phản xạ của ánh sáng  V (được gọi là vector view) là một vector từ pixel đến mắt |

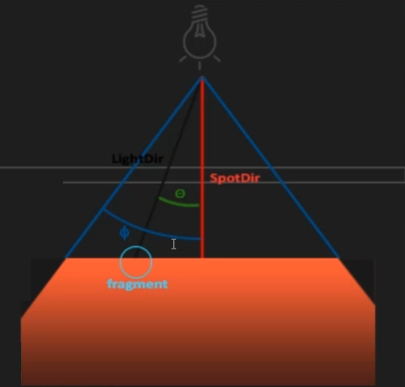
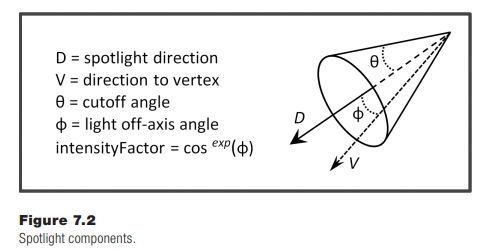
* 1. Các loại ánh sáng
     1. Global – Ánh sáng môi trường
        + Đơn giản nhất để làm mô hình.
        + Không có vị trí nguồn ánh sáng.
        + Ánh sáng bằng nhau ở mọi nơi. Tại mỗi pixel trên mọi vật thể trong scene
        + Ánh sáng không có sự khuếch tán hoặc specular. Nó chỉ định dưới dạng RGBA.
        + Công thức: float globalAmbient[4] = { 0.6f, 0.6f, 0.6f, 1.0f };
     2. Directional( hoặc Distance)
        + Ánh sáng cũng không có vị trí nguồn, nhưng nó có phương hướng. Nó hữu ích cho các tình huống mà nguồn ánh sáng ở xa xa coi như tía sáng chiếu song song song,ví dụ: ánh sáng mặt trời.
        + Trong nhiều tình huống như vậy, chúng tôi có thể chỉ quan tâm đến việc lập mô hình ánh sáng và không phải là vật tạo ra ánh sáng.
        + Ánh sáng này ảnh hưởng tới vật thể dựa vào góc tác động của ánh sáng; các vật sáng hơn ở phía đối diện với ánh sáng định hướng hơn là ở phía tiếp tuyến hoặc đối diện.
        + Mô hình ánh sáng này cần phải có đầu vào là các hướng của loại ánh sáng.
     3. Positional (hoặc point source)
        + Ánh sáng có vị trí cụ thể trong scene.Ví dụ: đèn , nến,… ánh sáng phát ra từ 1 điểm.
        + Vật thể chịu tác động của ánh sáng dựa vào góc tác động.Tuy nhiên hướng không được chỉ định nó khác nhau đối với mỗi đỉnh trong scene.
        + Cường độ của ánh sáng phụ thuộc vào khoảng cách từ ánh sáng đến vật thể, càng xa thì càng mờ.
        + Các hệ số suy giảm được mô hình hóa theo nhiều cách khác nhau. Dưới đây là công thức xác định sự suy giảm ánh sáng theo khoảng cách d.



* + - * Trong đó: Kc,Kt,Kq: tham số không âm của contant, linear và quadratic; D: khoảng cách đến nguồn sáng; Nhân hệ số attenuatationFactor với cường đồ ánh sáng sẽ trả về sự giảm cường độ theo khoảng cách đến nguồn sáng; kc,kl,kq= [0…1] do người dùng nhập vào.Một số bảng tra ví dụ:



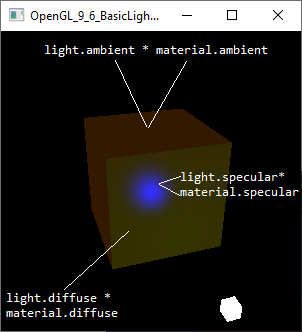
* + 1. Spotlight
       - Có cả direct và position light.



* + - * Ánh sáng này có dạng hình nón chiều rộng phi của nón giao động từ 0 đến 90 dg.
  1. Code tính toán các loại ánh sáng
     1. ASD cơ bản

|  |
| --- |
| // file frag  #version 430  out vec4 FragColor;  in vec2 textCoord;  in vec3 normal;  in vec3 modelPos;  struct Material{  vec3 ambient;  vec3 diffuse;  vec3 specular;  float shininess;  };  struct Light {  vec3 ambient;  vec3 diffuse;  vec3 specular;  vec3 lightColor;  vec3 lightPos;  };  uniform Material material;  uniform Light light;  uniform vec3 viewPos;  void main(void)  {  // Cường độ sáng  vec3 ambient= light.ambient \* material.ambient ;  // diffuse  vec3 norm = normalize(normal);  vec3 lightDir = normalize(light.lightPos-modelPos);  float diff = max(dot(norm,lightDir),0.0f);  vec3 diffuse =light.diffuse\*(material.diffuse \* diff);  //specular  vec3 viewDir= normalize(viewPos - modelPos);  vec3 reflectDir = reflect(-lightDir,norm);  float spec = pow(max(dot(viewDir,reflectDir),0.0f),material.shininess);  vec3 specular = light.specular\*(spec \*material.specular);  // combine  vec3 result = ambient + diffuse + specular;  FragColor = vec4(result, 1.0f);  } |

Kết quả



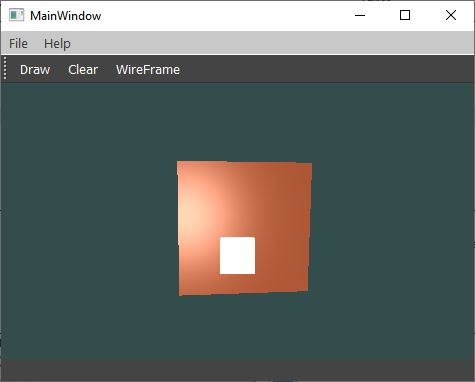
* + 1. Tính toán các loại ánh sáng(có chứa map)

|  |
| --- |
| // file vert  #version 430  layout (location = 0) in vec3 aPos;  layout (location=1) in vec3 aColor;  layout (location=2) in vec2 aTextCoord;  layout (location=3) in vec3 aNormal;  uniform mat4 projection;  uniform mat4 model;  uniform mat4 view;  out vec2 textCoord;  out vec3 normal;  out vec3 modelPos;  void main()  {  textCoord= aTextCoord;  normal= aNormal; // Normal = mat3(transpose(inverse(model))) \* aNormal;  modelPos = (model \* vec4(aPos,1.0f)).xyz ;  gl\_Position= projection \*view\*model\* vec4(aPos,1.0f);  } |
| // file frag  #version 430  out vec4 FragColor;  in vec2 textCoord;  in vec3 normal;  in vec3 modelPos;  struct Material{  vec3 ambient;  vec3 diffuse; // sampler2D diffuse;  vec3 specular; // sampler2D specular;  float shininess;  };  struct DirLight{  vec3 direction;  vec3 ambient;  vec3 diffuse;  vec3 specular;  };  struct PointLight{  vec3 lightPos;  vec3 ambient;  vec3 diffuse;  vec3 specular;  float quadratic;  float linear;  float constant;  };  // SpotLight= PointLight + DirLight  struct SpotLight{  vec3 lightPos;  vec3 direction;  vec3 ambient;  vec3 diffuse;  vec3 specular;    float quadratic;  float linear;  float constant;  float cutOff;  float outerCutOff;  };  uniform Material material;  uniform DirLight dirLight;  uniform PointLight pointLights;  uniform SpotLight spotLight;  uniform vec3 viewPos;  uniform bool useBlinnPhong;  uniform bool useGama;  uniform bool noTex;  vec3 calDirLight(DirLight light, vec3 norm, vec3 viewDir);  vec3 calPointLight(PointLight light, vec3 norm,vec3 modelPos, vec3 viewDir);  vec3 calSpotLight(SpotLight light, vec3 norm,vec3 modelPos, vec3 viewDir);  void main(void)  {  vec3 norm = normalize(normal);  vec3 viewDir= normalize(viewPos - modelPos);  // Tính toán lại diffuse và specular  vec3 ambientMap;  vec3 diffuseMap;  vec3 specularMap;  if(noTex){  // nếu material  // {…  // vec3 diffuse;  // vec3 specular; ..}  ambientMap= material.ambient;  diffuseMap= material.diffuse;  specularMap= material.specular;  }else{  // nếu material  // {…  // sampler2D diffuse;  // sampler2D specular; ..}  ambientMap= texture(material.diffuse, textCoord).xyz;  diffuseMap= texture(material.diffuse, textCoord).xyz;  specularMap= texture(material.specular, textCoord).xyz;  }  vec3 result = vec3(0,0,0);  // … Tính toán result  if (useGama){  float gama= 2.2;  result = pow(result, vec3(1.0/gama)) ;  }  FragColor = vec4(result, 1.0f);  }  vec3 calDirLight(DirLight light, vec3 norm, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.direction-modelPos); //cách 2: vec3 lightDir = normalize(-dirLight.direction);  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 ambient= light.ambient \* ambientMap;  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }    return ambient + diffuse + specular;  }  vec3 calPointLight(PointLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.lightPos-modelPos);  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 ambient= light.ambient \* ambientMap;  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }  // attenuation  float Distance = length(light.lightPos - modelPos);  float attenuation = 1.0f/ (light.constant + light.linear\*Distance + light.quadratic \*Distance);  ambient \*= attenuation;  diffuse \*= attenuation;  specular \*= attenuation;  return ambient + diffuse + specular;  }  vec3 calSpotLight(SpotLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.lightPos-modelPos);  float theta= dot(lightDir, normalize(-light.direction));  vec3 ambient = light.ambient \* diffuseMap;  if(theta >spotLight.outerCutOff){ // > because using cosines and not degrees  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }  // attenuation  float Distance = length(light.lightPos - modelPos);  float attenuation = 1.0f/ (light.constant + light.linear\*Distance + light.quadratic \*Distance);  // soft Edges  float epsilon = light.cutOff- light.outerCutOff;  float intensity = clamp((theta-light.outerCutOff)/epsilon,0.0f,1.0f);  ambient \*= attenuation\*intensity;  diffuse \*= attenuation\*intensity;  specular \*= attenuation\*intensity;  return ambient + diffuse + specular;  } else{  // render just ambient  return ambient;  }  } |

* 1. Ví dụ minh hoạ
     1. Ví dụ 1:ASD cơ bản

|  |
| --- |
| QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.2f, 0.2f, 0.2f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  //model.setToIdentity();  model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  // light properties  // lightingShader.setFloat("light.constant", 1.0f);  // lightingShader.setFloat("light.linear", 0.09f);  // lightingShader.setFloat("light.quadratic", 0.032f);  // lightingShader.setFloat("material.shininess", 32.0f);  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.diffuse", material\_diffuse);  shaderProgram.setUniformValue("material.specular", material\_specular);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  lightshaderProgram.bind();  modelLight.setToIdentity();  modelLight.scale(0.2f);  modelLight.translate(light\_pos.x(), light\_pos.y(), light\_pos.z());  lightshaderProgram.setUniformValue("projection", projection);  lightshaderProgram.setUniformValue("view", view);  lightshaderProgram.setUniformValue("model", modelLight);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  } |
| //light.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  out vec3 Normal;  out vec3 FragPos;  out vec2 TexCoords;  void main(){  Normal = mat3(transpose(inverse(model))) \* aNormal;  FragPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  TexCoords = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| //light.frag  #version 330 core  out vec4 FragColor;  void main(){  // Cường độ sáng  FragColor = vec4(1.0f,1.0f,1.0f, 1.0f);  }; |
| //obj.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  out vec3 Normal;  out vec3 FragPos;  out vec2 TexCoords;  void main(){  Normal = mat3(transpose(inverse(model))) \* aNormal;  FragPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  TexCoords = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| //obj.frag  #version 330 core  out vec4 FragColor;  struct Material{  vec3 ambient;  vec3 diffuse;  vec3 specular;  float shininess;  };  struct Light {  vec3 ambient;  vec3 diffuse;  vec3 specular;  vec3 lightColor;  vec3 lightPos;  };  uniform Material material;  uniform Light light;  uniform vec3 viewPos;  in vec3 Normal;  in vec3 FragPos;  in vec2 TexCoords;  void main(){  //FragColor = vec4(1.0f,0.0f,0.0f, 1.0f);  // Cường độ sáng  vec3 ambient= light.ambient \* material.ambient ;  // diffuse  vec3 norm = normalize(Normal);  vec3 lightDir = normalize(light.lightPos-FragPos);  float diff = max(dot(norm,lightDir),0.0f);  vec3 diffuse =light.diffuse\*(material.diffuse \* diff);  //specular  vec3 viewDir= normalize(viewPos - FragPos);  vec3 reflectDir = reflect(-lightDir,norm);  float spec = pow(max(dot(viewDir,reflectDir),0.0f),material.shininess);  vec3 specular = light.specular\*(spec \*material.specular);  // combine  vec3 result = ambient + diffuse + specular;  FragColor = vec4(result, 1.0f);  }; |

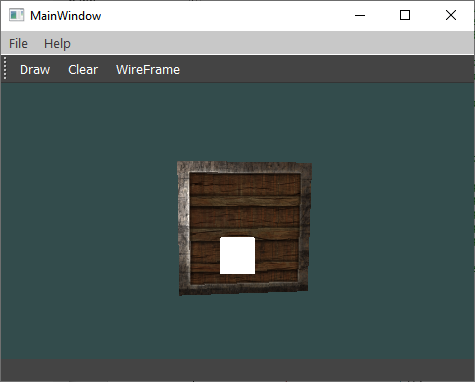
Kết quả



* + 1. Ví dụ 2:Specular map

|  |
| --- |
| shaderProgram.bind();  // Texture  texDiffuse = new QOpenGLTexture(QImage(":/Images/Resources/Images/diffuseMapTex.png").mirrored());  texSpecular = new QOpenGLTexture(QImage(":/Images/Resources/Images/specularMapTex.png").mirrored());  // set cổng trong shader  shaderProgram.setUniformValue("material.diffuse",0);  shaderProgram.setUniformValue("material.specular", 1); |
| QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.2f, 0.2f, 0.2f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  //model.setToIdentity();  model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  // light properties  // lightingShader.setFloat("light.constant", 1.0f);  // lightingShader.setFloat("light.linear", 0.09f);  // lightingShader.setFloat("light.quadratic", 0.032f);  // lightingShader.setFloat("material.shininess", 32.0f);  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  // shaderProgram.setUniformValue("material.diffuse", material\_diffuse);  // shaderProgram.setUniformValue("material.specular", material\_specular);  texDiffuse->bind(0);  texSpecular->bind(1);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  //  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  lightshaderProgram.bind();  modelLight.setToIdentity();  modelLight.scale(0.2f);  modelLight.translate(light\_pos.x(), light\_pos.y(), light\_pos.z());  lightshaderProgram.setUniformValue("projection", projection);  lightshaderProgram.setUniformValue("view", view);  lightshaderProgram.setUniformValue("model", modelLight);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  } |

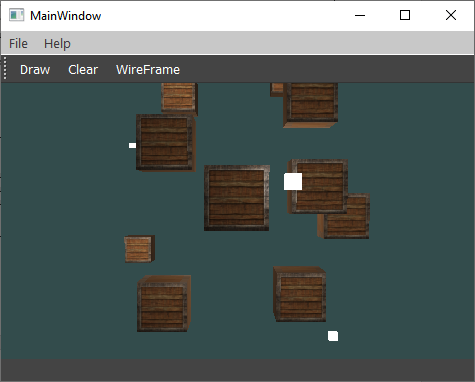
Kết quả



* + 1. Ví dụ 3:Kết hợp nhiều loại ánh sáng

|  |
| --- |
| void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  // material  shaderProgram.setUniformValue("material.ambient", 1.0f, 0.5f, 0.31f);  texDiffuse->bind(0);  texSpecular->bind(1);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // directional light  shaderProgram.setUniformValue("dirLight.direction", -0.2f, -1.0f, -0.3f);  shaderProgram.setUniformValue("dirLight.ambient", 0.05f, 0.05f, 0.05f);  shaderProgram.setUniformValue("dirLight.diffuse", 0.4f, 0.4f, 0.4f);  shaderProgram.setUniformValue("dirLight.specular", 0.5f, 0.5f, 0.5f);  // point light 1  shaderProgram.setUniformValue("pointLights[0].lightPos", pointLightPositions[0]);  shaderProgram.setUniformValue("pointLights[0].ambient", 0.05f, 0.05f, 0.05f);  shaderProgram.setUniformValue("pointLights[0].diffuse", 0.8f, 0.8f, 0.8f);  shaderProgram.setUniformValue("pointLights[0].specular", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("pointLights[0].quadratic", 0.032f);  shaderProgram.setUniformValue("pointLights[0].linear", 0.09f);  shaderProgram.setUniformValue("pointLights[0].constant", 1.0f);  // point light 2  shaderProgram.setUniformValue("pointLights[1].lightPos", pointLightPositions[1]);  shaderProgram.setUniformValue("pointLights[1].ambient", 0.05f, 0.05f, 0.05f);  shaderProgram.setUniformValue("pointLights[1].diffuse", 0.8f, 0.8f, 0.8f);  shaderProgram.setUniformValue("pointLights[1].specular", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("pointLights[1].quadratic", 0.032f);  shaderProgram.setUniformValue("pointLights[1].linear", 0.09f);  shaderProgram.setUniformValue("pointLights[1].constant", 1.0f);  // point light 3  shaderProgram.setUniformValue("pointLights[2].lightPos", pointLightPositions[2]);  shaderProgram.setUniformValue("pointLights[2].ambient", 0.05f, 0.05f, 0.05f);  shaderProgram.setUniformValue("pointLights[2].diffuse", 0.8f, 0.8f, 0.8f);  shaderProgram.setUniformValue("pointLights[2].specular", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("pointLights[2].quadratic", 0.032f);  shaderProgram.setUniformValue("pointLights[2].linear", 0.09f);  shaderProgram.setUniformValue("pointLights[2].constant", 1.0f);  // point light 4  shaderProgram.setUniformValue("pointLights[3].lightPos", pointLightPositions[3]);  shaderProgram.setUniformValue("pointLights[3].ambient", 0.05f, 0.05f, 0.05f);  shaderProgram.setUniformValue("pointLights[3].diffuse", 0.8f, 0.8f, 0.8f);  shaderProgram.setUniformValue("pointLights[3].specular", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("pointLights[3].quadratic", 0.032f);  shaderProgram.setUniformValue("pointLights[3].linear", 0.09f);  shaderProgram.setUniformValue("pointLights[3].constant", 1.0f);  // spotLight  shaderProgram.setUniformValue("spotLight.lightPos", pointLightPositions[3]);  shaderProgram.setUniformValue("spotLight.direction", m\_camera.Front);  shaderProgram.setUniformValue("spotLight.ambient", 0.0f, 0.0f, 0.0f);  shaderProgram.setUniformValue("spotLight.diffuse", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("spotLight.specular", 1.0f, 1.0f, 1.0f);  shaderProgram.setUniformValue("spotLight.constant", 1.0f);  shaderProgram.setUniformValue("spotLight.linear", 0.09f);  shaderProgram.setUniformValue("spotLight.quadratic", 0.032f);  float cutoffvalue = cos(12.5f \*3.14159/180);  shaderProgram.setUniformValue("spotLight.cutOff",cutoffvalue );  float outercutoffvalue = cos(15.0f \*3.14159/180);  shaderProgram.setUniformValue("spotLight.outerCutOff", outercutoffvalue);  // draw cubes  foreach (QVector3D pos, cubePositions) {  model.setToIdentity();  model.translate(pos);  shaderProgram.setUniformValue("model", model);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  }  // draw point lights  lightshaderProgram.bind();  lightshaderProgram.setUniformValue("projection", projection);  lightshaderProgram.setUniformValue("view", view);  foreach (QVector3D pos, pointLightPositions) {  modelLight.setToIdentity();  modelLight.translate(pos);  modelLight.scale(0.2f);  lightshaderProgram.setUniformValue("model", modelLight);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 36);  }  } |
| // light.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  out vec3 Normal;  out vec3 FragPos;  out vec2 TexCoords;  void main(){  Normal = mat3(transpose(inverse(model))) \* aNormal;  FragPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  TexCoords = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| // light.frag  #version 330 core  out vec4 FragColor;  void main(){  // Cường độ sáng  FragColor = vec4(1.0f,1.0f,1.0f, 1.0f);  }; |
| // obj.frag  #version 330  out vec4 FragColor;  in vec2 textCoord;  in vec3 normal;  in vec3 modelPos;  struct Material{  vec3 ambient;  sampler2D diffuse; // vec3 diffuse;  sampler2D specular;// vec3 specular;  float shininess;  };  struct DirLight{  vec3 direction;  vec3 ambient;  vec3 diffuse;  vec3 specular;  };  struct PointLight{  vec3 lightPos;  vec3 ambient;  vec3 diffuse;  vec3 specular;  float quadratic;  float linear;  float constant;  };  // SpotLight= PointLight + DirLight  struct SpotLight{  vec3 lightPos;  vec3 direction;  vec3 ambient;  vec3 diffuse;  vec3 specular;  float quadratic;  float linear;  float constant;  float cutOff;  float outerCutOff;  };  #define NR\_POINT\_LIGHTS 4  uniform Material material;  uniform DirLight dirLight;  uniform PointLight pointLights[NR\_POINT\_LIGHTS];  uniform SpotLight spotLight;  uniform vec3 viewPos;  uniform bool useBlinnPhong;  uniform bool useGama;  //uniform bool noTex;  vec3 calDirLight(DirLight light, vec3 norm, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap);  vec3 calPointLight(PointLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap);  vec3 calSpotLight(SpotLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap);  void main(void)  {  vec3 norm = normalize(normal);  vec3 viewDir= normalize(viewPos - modelPos);  // Tính toán lại diffuse và specular  vec3 ambientMap;  vec3 diffuseMap;  vec3 specularMap;  bool noTex= false;  // if(noTex){  // nếu material  // {…  // vec3 diffuse;  // vec3 specular; ..}  // ambientMap= material.ambient;  // diffuseMap= material.diffuse;  // specularMap= material.specular;  //}else{  // nếu material  // {…  // sampler2D diffuse;  // sampler2D specular; ..}  ambientMap= texture(material.diffuse, textCoord).xyz;  diffuseMap= texture(material.diffuse, textCoord).xyz;  specularMap= texture(material.specular, textCoord).xyz;  //}  vec3 result = vec3(0,0,0);  // … Tính toán result  // if (useGama){  // float gama= 2.2;  // result = pow(result, vec3(1.0/gama)) ;  // }  // combine multiphe lights  // phase 1: directional lighting  result += calDirLight(dirLight, norm, viewDir,ambientMap, diffuseMap,specularMap);  // phase 2: point lights  for(int i = 0; i < NR\_POINT\_LIGHTS; i++)  result += calPointLight(pointLights[i], norm, modelPos, viewDir,ambientMap, diffuseMap,specularMap);  // // phase 3: spot light  result += calSpotLight(spotLight, norm, modelPos, viewDir, ambientMap, diffuseMap,specularMap);  FragColor = vec4(result, 1.0f);  }  vec3 calDirLight(DirLight light, vec3 norm, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.direction-modelPos); //cách 2: vec3 lightDir = normalize(-dirLight.direction);  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 ambient= light.ambient \* ambientMap;  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }  return ambient + diffuse + specular;  }  vec3 calPointLight(PointLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.lightPos-modelPos);  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 ambient= light.ambient \* ambientMap;  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }  // attenuation  float Distance = length(light.lightPos - modelPos);  float attenuation = 1.0f/ (light.constant + light.linear\*Distance + light.quadratic \*Distance);  ambient \*= attenuation;  diffuse \*= attenuation;  specular \*= attenuation;  return ambient + diffuse + specular;  }  vec3 calSpotLight(SpotLight light, vec3 norm,vec3 modelPos, vec3 viewDir, vec3 ambientMap, vec3 diffuseMap, vec3 specularMap ){  vec3 lightDir = normalize(light.lightPos-modelPos);  float theta= dot(lightDir, normalize(-light.direction));  vec3 ambient = light.ambient \* diffuseMap;  if(theta >spotLight.outerCutOff){ // > because using cosines and not degrees  //Diffuse shading  float diff = max(dot(norm,lightDir),0.0f);  // combine result  vec3 diffuse=light.diffuse\*( diffuseMap \* diff);  // specular shading  vec3 specular= vec3(0,0,0);  if (diff >0){  float dotProd =0.0;  if (useBlinnPhong){  // calculate using Blinn-Phong model  vec3 halfwaydir = normalize(lightDir + viewDir);  dotProd = dot(norm,halfwaydir);  }else{  // calculate using Phong model  vec3 reflectDir = reflect(-lightDir,norm);  dotProd = dot(viewDir,reflectDir);  }  float spec = pow(max(dotProd,0.0f),material.shininess);  specular = light.specular\*(spec \* specularMap );  }  // attenuation  float Distance = length(light.lightPos - modelPos);  float attenuation = 1.0f/ (light.constant + light.linear\*Distance + light.quadratic \*Distance);  // soft Edges  float epsilon = light.cutOff- light.outerCutOff;  float intensity = clamp((theta-light.outerCutOff)/epsilon,0.0f,1.0f);  ambient \*= attenuation\*intensity;  diffuse \*= attenuation\*intensity;  specular \*= attenuation\*intensity;  return ambient + diffuse + specular;  } else{  // render just ambient  return ambient;  }  } |
| // obj.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  //out vec3 Normal;  //out vec3 FragPos;  //out vec2 TexCoords;  out vec2 textCoord;  out vec3 normal;  out vec3 modelPos;  void main(){  normal = mat3(transpose(inverse(model))) \* aNormal;  modelPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  textCoord = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |

Kết quả



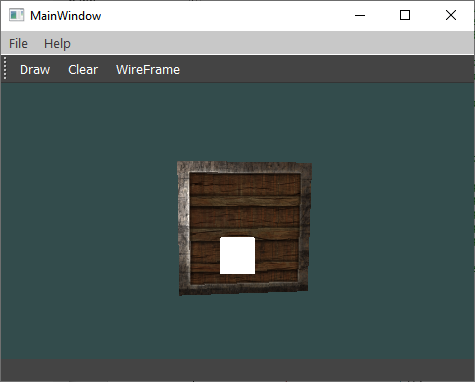
1. Assimp(Load model – Qt: 5.9 – MSCV2015 – Assim:5.0.1)
   1. Mesh
      1. Lý thuyết
      2. Thực hành
         * Tạo class mesh

|  |
| --- |
| #ifndef MESH\_H  #define MESH\_H  #include <QVector>  #include <QOpenGLFunctions\_3\_3\_Core>  #include <QOpenGLShaderProgram>  #include <vector>  #include <QVector3D>  #include <QVector2D>  using namespace std;  struct **Vertex**{  QVector3D Position;  QVector3D Normal;  QVector2D TexCoords;  };  struct **Texture**{  unsigned int id;  string type;  string path;  };  class **Mesh**  {  public:  // Mesh data  vector<Vertex> verties;  vector<unsigned int> indices;  vector<Texture> textures;  void **Draw**(QOpenGLShaderProgram &shader);  **Mesh**(QOpenGLFunctions\_3\_3\_Core \*glFuns,  vector<Vertex> vertives, vector<unsigned int> indices,  vector<Texture> textures);  private:  unsigned int VAO,VBO, EBO;  void **setupMesh**();  QOpenGLFunctions\_3\_3\_Core \*m\_glFuns;  };  #endif // MESH\_H |
| #include "mesh.h"  Mesh::**Mesh**(QOpenGLFunctions\_3\_3\_Core \*glFuns, vector<Vertex> vertives, vector<unsigned int> indices, vector<Texture> textures)  {  this->verties= vertives;  this->indices= indices;  this->textures = textures;  this->m\_glFuns= glFuns;  setupMesh();  }  void Mesh::**setupMesh**()  {  m\_glFuns->glGenVertexArrays(1,&VAO);  m\_glFuns->glGenBuffers(1,&VBO);  m\_glFuns->glGenBuffers(1,&EBO);  m\_glFuns->glBindVertexArray(VAO);  m\_glFuns->glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  m\_glFuns->glBufferData(GL\_ARRAY\_BUFFER, verties.size()\*sizeof (Vertex), &verties[0], GL\_STATIC\_DRAW);  m\_glFuns->glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  m\_glFuns->glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size()\*sizeof (unsigned int), &indices[0], GL\_STATIC\_DRAW);  // Set atribute  m\_glFuns->glVertexAttribPointer(0,3,GL\_FLOAT, GL\_FALSE,8\*sizeof (float),(void\*)0);  m\_glFuns->glEnableVertexAttribArray(0);  // m\_glFuns->glVertexAttribPointer(1,3,GL\_FLOAT, GL\_FALSE,8\*sizeof (float),(void\*)(3\*sizeof (float)));  // m\_glFuns->glEnableVertexAttribArray(1);  // m\_glFuns->glVertexAttribPointer(2,2,GL\_FLOAT, GL\_FALSE,8\*sizeof (float),(void\*)(6\*sizeof (float)));  // m\_glFuns->glEnableVertexAttribArray(2);  m\_glFuns->glVertexAttribPointer(1,3,GL\_FLOAT, GL\_FALSE,sizeof (Vertex),(void\*)offsetof(Vertex,Normal));  m\_glFuns->glEnableVertexAttribArray(1);  m\_glFuns->glVertexAttribPointer(2,2,GL\_FLOAT, GL\_FALSE,sizeof (Vertex),(void\*)offsetof(Vertex,TexCoords));  m\_glFuns->glEnableVertexAttribArray(2);  //Giai phong  // m\_glFuns->glBindBuffer(GL\_ARRAY\_BUFFER, 0); // Không cần đống EBO.Dữ liệu tự nén vào khi kết thúc VAO  // m\_glFuns->glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0); // không cần đóng VBO. Dữ liệu tự nén vào khi kết thúc VAO  m\_glFuns->glBindVertexArray(0);  }  void Mesh::**Draw**(QOpenGLShaderProgram &shader)  {  unsigned int diffuseNr =1;  unsigned int specularNr =1;  for(unsigned int i=0; i< textures.size(); i++){  m\_glFuns->glActiveTexture(GL\_TEXTURE0+i);  string number;  string name=textures[i].type;  if (name== "texture\_diffuse"){  number = std::to\_string(diffuseNr++);  }else if(name == "texture\_specular"){  number= std::to\_string(specularNr++);  }  shader.setUniformValue(("material."+name+number).c\_str(),i);  m\_glFuns->glBindTexture(GL\_TEXTURE\_2D,textures[i].id);  }  m\_glFuns->glBindVertexArray(VAO);  //m\_glFuns->glDrawArrays(GL\_TRIANGLES,0,verties.size());  m\_glFuns->glDrawElements(GL\_TRIANGLES,verties.size(),GL\_UNSIGNED\_INT, 0);  } |

* + - * Sử dụng mesh trong project

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include "camera.h"  #include <QPointF>  #include <QOpenGLShaderProgram>  #include <QTimer>  #include <QOpenGLTexture>  #include <mesh.h>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit **AXBOpenGLWidget**(QWidget \*parent = nullptr);  private slots:  void **on\_TimeOut**();  protected:  void ***initializeGL***();  void ***resizeGL***(int w, int h);  void ***paintGL***();  // Event  void ***mousePressEvent***(QMouseEvent \*event);  void ***mouseMoveEvent***(QMouseEvent \*event);  void ***wheelEvent***(QWheelEvent \*event);  void ***keyPressEvent***(QKeyEvent \*event);  private:  Camera m\_camera;  QOpenGLShaderProgram shaderProgram, lightshaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model,modelLight;  QOpenGLTexture \*texDiffuse;  QOpenGLTexture \*texSpecular;  Mesh \*m\_mesh;  Mesh \* **processMesh**();  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QMouseEvent>  #include <QWheelEvent>  #include <QKeyEvent>  #include <QDebug>  QPoint lastPos(0,0);  int perFrame = 100;// delta time / frame  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget(parent)  {  ...  }  float vertices[] = {  // positions // normals // texture coords  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f,  ...  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/light.vert");  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/light.frag");  lightshaderProgram.*link*();  success = lightshaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << lightshaderProgram.log();  // Texture  texDiffuse = new QOpenGLTexture(QImage(":/Images/Resources/Images/diffuseMapTex.png").mirrored());  texSpecular = new QOpenGLTexture(QImage(":/Images/Resources/Images/specularMapTex.png").mirrored());  // Sử dụng mesh  m\_mesh= processMesh();  }  Mesh\* AXBOpenGLWidget::**processMesh**()  {  vector<Vertex> \_vertices;  vector<unsigned int> \_indices;  vector<Texture> \_textures;  // ==== Cach 1============  for(int i=0; i<36; i++){  Vertex vert;  vert.Position[0]=vertices[i\*8+0];  vert.Position[1]=vertices[i\*8+1];  vert.Position[2]=vertices[i\*8+2];  vert.Normal[0]=vertices[i\*8+3];  vert.Normal[1]=vertices[i\*8+4];  vert.Normal[2]=vertices[i\*8+5];  vert.TexCoords[0]=vertices[i\*8+6];  vert.TexCoords[1]=vertices[i\*8+7];  \_vertices.push\_back(vert);  \_indices.push\_back(i);  }  // --- cach 2---------  // memcmp(&\_vertices[0],vertices,sizeof (vertices));  // for(int i=0; i<36; i++){  // \_indices.push\_back(i);  // }  Texture tex;  tex.id = texDiffuse->textureId();  tex.type ="texture\_diffuse"; // material.texture\_diffuse trong shader  \_textures.push\_back(tex);  tex.id = texSpecular->textureId();  tex.type ="texture\_specular"; // material.texture\_specular trong shader  \_textures.push\_back(tex);  return new Mesh(  QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(),  \_vertices, \_indices, \_textures  );  }  QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  //QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  //QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.2f, 0.2f, 0.2f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  //model.setToIdentity();  model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  m\_mesh->Draw(*shaderProgram*); // sử dụng mesh để draw  lightshaderProgram.bind();  modelLight.setToIdentity();  modelLight.scale(0.2f);  modelLight.translate(light\_pos);  lightshaderProgram.setUniformValue("projection", projection);  lightshaderProgram.setUniformValue("view", view);  lightshaderProgram.setUniformValue("model", modelLight);  m\_mesh->Draw(*lightshaderProgram*); // sử dụng mesh để draw  } |
| // light.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  out vec3 Normal;  out vec3 FragPos;  out vec2 TexCoords;  void main(){  Normal = mat3(transpose(inverse(model))) \* aNormal;  FragPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  TexCoords = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| // light.frag  #version 330 core  out vec4 FragColor;  void main(){  // Cường độ sáng  FragColor = vec4(1.0f,1.0f,1.0f, 1.0f);  }; |
| // obj.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 1) in vec3 aNormal;  layout (location = 2) in vec2 aTexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  out vec3 Normal;  out vec3 FragPos;  out vec2 TexCoords;  void main(){  Normal = mat3(transpose(inverse(model))) \* aNormal;  FragPos=vec3(model \* vec4(aPos.x,aPos.y,aPos.z,1.0)) ;  TexCoords = aTexCoords;  gl\_Position = projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  }; |
| // obj.frag  #version 330 core  out vec4 FragColor;  struct Material{  vec3 ambient;  sampler2D texture\_diffuse1;  sampler2D texture\_specular1;  float shininess;  };  struct Light {  vec3 ambient;  vec3 diffuse;  vec3 specular;  vec3 lightColor;  vec3 lightPos;  };  uniform Material material;  uniform Light light;  uniform vec3 viewPos;  in vec3 Normal;  in vec3 FragPos;  in vec2 TexCoords;  void main(){  FragColor = vec4(1.0f,0.0f,0.0f, 1.0f);  vec3 ambientMap= texture(material.texture\_diffuse1, TexCoords).xyz;  vec3 diffuseMap= texture(material.texture\_diffuse1, TexCoords).xyz;  vec3 specularMap= texture(material.texture\_specular1, TexCoords).xyz;  // Cường độ sáng  vec3 ambient= light.ambient \* ambientMap ;  // diffuse  vec3 norm = normalize(Normal);  vec3 lightDir = normalize(light.lightPos-FragPos);  float diff = max(dot(norm,lightDir),0.0f);  vec3 diffuse =light.diffuse\*(diffuseMap \* diff);  //specular  vec3 viewDir= normalize(viewPos - FragPos);  vec3 reflectDir = reflect(-lightDir,norm);  float spec = pow(max(dot(viewDir,reflectDir),0.0f),material.shininess);  vec3 specular = light.specular\*(spec \*specularMap);  // combine  vec3 result = ambient + diffuse + specular;  FragColor = vec4(result, 1.0f);  }; |

Kết quả



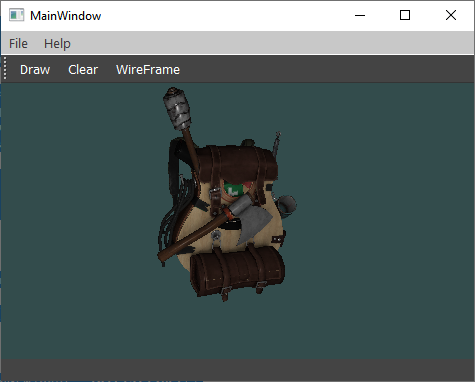
* 1. Load Model
     1. Lý thuyết
     2. Thực hành
        + Tạo class Model

|  |
| --- |
| #ifndef MODEL\_H  #define MODEL\_H  #include "mesh.h"  #include <assimp/Importer.hpp>  #include <assimp/scene.h>  #include <assimp/postprocess.h>  #include <QOpenGLShaderProgram>  #include <QOpenGLFunctions\_3\_3\_Core>  class **Model**  {  public:  vector<Texture> textures\_loaded;  **Model**(QOpenGLFunctions\_3\_3\_Core \*glFuncs,const char \*path){  this->glFuncs= glFuncs;  loadModel(path);  };  void **Draw**(QOpenGLShaderProgram &shader){  for(unsigned int i=0; i <meshes.size(); i++){  meshes[i].Draw(*shader*);  }  }  float m\_minX=100, m\_maxX=-100, m\_minY=100, m\_maxY=0, m\_minZ=100, m\_maxZ=-100;  private:  QOpenGLFunctions\_3\_3\_Core \*glFuncs;  vector<Mesh> meshes;  string directory;  void **loadModel**(string path);  void **processNode**(aiNode \*node, const aiScene \*scene);  Mesh **processMesh**(aiMesh \*mesh, const aiScene \*scene);  vector<Texture> **loadMaterialTextures**(aiMaterial \*mat, aiTextureType type, string typeName);  unsigned int **TextureFromFile**(const char \*path, const string &directory);  };  #endif // MODEL\_H |
| #include "model.h"  using namespace std;  #include <QDebug>  #include <QOpenGLTexture>  void Model::**loadModel**(string path)  {  Assimp::Importer import;  const aiScene \*scene = import.ReadFile(path,aiProcess\_Triangulate | aiProcess\_FlipUVs);  if(!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode){  qDebug() << "ERROR::ASSIMP::" << import.GetErrorString()<< endl;  return;  }  directory = path.substr(0,path.find\_last\_of('/'));  processNode(scene->mRootNode,scene);  }  void Model::**processNode**(aiNode \*node, const aiScene \*scene)  {  // process all the node's meshes (if any)  for(unsigned int i=0; i< node->mNumMeshes; i++){  aiMesh \*mesh = scene->mMeshes[node->mMeshes[i]];  meshes.push\_back(processMesh(mesh,scene));  }  // the do the same for each of its children  for(unsigned int i=0; i< node->mNumChildren; i++){  processNode(node->mChildren[i],scene);  }  }  Mesh Model::**processMesh**(aiMesh \*mesh, const aiScene \*scene)  {  vector<Vertex> vertices;  vector<unsigned int> indices;  vector<Texture> textures;  // walk through each of the mesh's vertices  for(unsigned int i = 0; i < mesh->mNumVertices; i++)  {  Vertex vertex;  QVector3D vector; // we declare a placeholder vector since assimp uses its own vector class that doesn't directly convert to glm's vec3 class so we transfer the data to this placeholder glm::vec3 first.  // tính toán điểm min và max  if(m\_minX> mesh->mVertices[i].x) m\_minX= mesh->mVertices[i].x;  if(m\_maxX<mesh->mVertices[i].x) m\_maxX= mesh->mVertices[i].x;  if(m\_minY>mesh->mVertices[i].y) m\_minY= mesh->mVertices[i].y;  if(m\_maxY<mesh->mVertices[i].y) m\_maxY= mesh->mVertices[i].y;  if(m\_minZ>mesh->mVertices[i].z) m\_minZ= mesh->mVertices[i].z;  if(m\_maxZ<mesh->mVertices[i].z) m\_maxZ= mesh->mVertices[i].z;  // positions  vector.setX(mesh->mVertices[i].x);  vector.setY(mesh->mVertices[i].y);  vector.setZ(mesh->mVertices[i].z);  vertex.Position = vector;  // normals  if (mesh->HasNormals())  {  vector.setX( mesh->mNormals[i].x);  vector.setY( mesh->mNormals[i].y);  vector.setZ( mesh->mNormals[i].z);  vertex.Normal = vector;  }  // texture coordinates  if(mesh->mTextureCoords[0]) // does the mesh contain texture coordinates?  {  QVector2D vec;  // a vertex can contain up to 8 different texture coordinates. We thus make the assumption that we won't  // use models where a vertex can have multiple texture coordinates so we always take the first set (0).  vec.setX( mesh->mTextureCoords[0][i].x);  vec.setY( mesh->mTextureCoords[0][i].y);  vertex.TexCoords = vec;  // tangent  // vector.x = mesh->mTangents[i].x;  // vector.y = mesh->mTangents[i].y;  // vector.z = mesh->mTangents[i].z;  // vertex.Tangent = vector;  // bitangent  // vector.x = mesh->mBitangents[i].x;  // vector.y = mesh->mBitangents[i].y;  // vector.z = mesh->mBitangents[i].z;  // vertex.Bitangent = vector;  }  else  vertex.TexCoords = QVector2D(0.0f, 0.0f);  vertices.push\_back(vertex);  }  // now wak through each of the mesh's faces (a face is a mesh its triangle) and retrieve the corresponding vertex indices.  for(unsigned int i = 0; i < mesh->mNumFaces; i++)  {  aiFace face = mesh->mFaces[i];  // retrieve all indices of the face and store them in the indices vector  for(unsigned int j = 0; j < face.mNumIndices; j++)  indices.push\_back(face.mIndices[j]);  }  // process materials  aiMaterial\* material = scene->mMaterials[mesh->mMaterialIndex];  // we assume a convention for sampler names in the shaders. Each diffuse texture should be named  // as 'texture\_diffuseN' where N is a sequential number ranging from 1 to MAX\_SAMPLER\_NUMBER.  // Same applies to other texture as the following list summarizes:  // diffuse: texture\_diffuseN  // specular: texture\_specularN  // normal: texture\_normalN  if (mesh->mMaterialIndex>=0){  // 1. diffuse maps  vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType\_DIFFUSE, "texture\_diffuse");  textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());  // 2. specular maps  vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType\_SPECULAR, "texture\_specular");  textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());  // 3. normal maps  std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType\_HEIGHT, "texture\_normal");  textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());  // 4. height maps  std::vector<Texture> heightMaps = loadMaterialTextures(material, aiTextureType\_AMBIENT, "texture\_height");  textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());  }  return Mesh(glFuncs,vertices,indices, textures);  }  vector<Texture> Model::**loadMaterialTextures**(aiMaterial \*mat, aiTextureType type, string typeName)  {  vector<Texture> textures;  for(unsigned int i=0; i< mat->GetTextureCount(type); i++){  aiString str;  mat->GetTexture(type, i, &str);  bool skip = false;  for(unsigned int j=0 ; j< textures\_loaded.size(); j++){  if(std::strcmp(textures\_loaded[j].path.data(), str.C\_Str())==0){  textures.push\_back(textures\_loaded[j]);  skip =true;  break;  }  }  if(!skip){  Texture texture;  texture.id = TextureFromFile(str.C\_Str(), directory);  texture.type= typeName;  texture.path = str.C\_Str();  textures.push\_back(texture);  textures\_loaded.push\_back(texture);  }  }  return textures;  }  unsigned int Model::**TextureFromFile**(const char \*path, const string &directory){  string fileName = string(path);  fileName = directory + '/' + fileName;  QOpenGLTexture \*texture = new QOpenGLTexture(QImage(fileName.c\_str()).mirrored());  if(texture == NULL) qDebug() << "texture is NULL";  else qDebug() << fileName.c\_str() << "loaded";  return texture->textureId();  } |

* + - * Sử dụng class model trong opengl widget

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include "camera.h"  #include <QPointF>  #include <QOpenGLShaderProgram>  #include <QTimer>  #include <QOpenGLTexture>  #include "model.h"  class AXBOpenGLWidget : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit AXBOpenGLWidget(QWidget \*parent = nullptr);  private slots:  void on\_TimeOut();  protected:  void initializeGL();  void resizeGL(int w, int h);  void paintGL();  // Event  void mousePressEvent(QMouseEvent \*event);  void mouseMoveEvent(QMouseEvent \*event);  void wheelEvent(QWheelEvent \*event);  void keyPressEvent(QKeyEvent \*event);  private:  Camera m\_camera;  QOpenGLShaderProgram shaderProgram, lightshaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model,modelLight;  QOpenGLTexture \*texDiffuse;  QOpenGLTexture \*texSpecular;  Model \* m\_model;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QMouseEvent>  #include <QWheelEvent>  #include <QKeyEvent>  #include <QDebug>  QPoint lastPos(0,0);  int perFrame = 100;// delta time / frame  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget(parent)  {  ...  }  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  ...  // Texture  texDiffuse = new QOpenGLTexture(QImage(":/Images/Resources/Images/diffuseMapTex.png").mirrored());  texSpecular = new QOpenGLTexture(QImage(":/Images/Resources/Images/specularMapTex.png").mirrored());  // Sử dụng mesh  m\_model= new Model(QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(),  "C:/Users/HUY NGUYEN/Desktop/OpenGL vs QT/OpenGLAndQT/Chuong 4\_Assimp/backpack/backpack.obj");  }  …  QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  //QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  //QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.2f, 0.2f, 0.2f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  //model.setToIdentity();  model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  m\_model->Draw(*shaderProgram*); // sử dụng model để draw  // lightshaderProgram.bind();  // modelLight.setToIdentity();  // modelLight.scale(0.2f);  // modelLight.translate(light\_pos);  // lightshaderProgram.setUniformValue("projection", projection);  // lightshaderProgram.setUniformValue("view", view);  // lightshaderProgram.setUniformValue("model", modelLight);  // m\_model->Draw(lightshaderProgram); // sử dụng mesh để draw  }  ... |

Kết quả



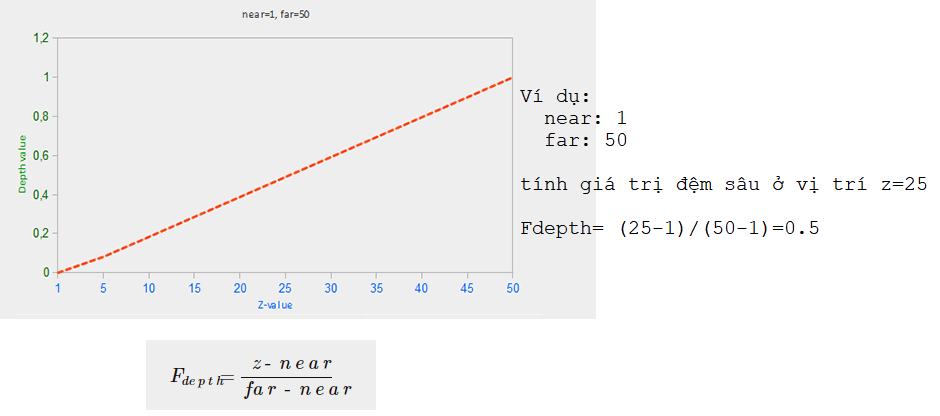
1. Depth Test
   1. Lý thuyết về depth test
      1. Depth Test
         * Với mỗi pixel trên màn hình, bộ đệm chiều sâu lưu khoảng cách vuông góc từ điểm nhìn đến pixel đó. Nên nếu giá trị chiều sâu của một điểm được ánh xạ vào pixel đó nhỏ hơn giá trị được lưu trong bộ đêm chiều sâu thì điểm này được coi là qua Depth test (depth test pass) và giá trị chiều sâu của nó được thay thế cho giá trị lưu trong bộ đệm. Nếu giá trị chiều sâu của điểm đó lớn hơn giá trị lưu trong Depth Buffer thì điểm đó “trượt” phép kiểm tra chiều sâu. (Depth test Fail)

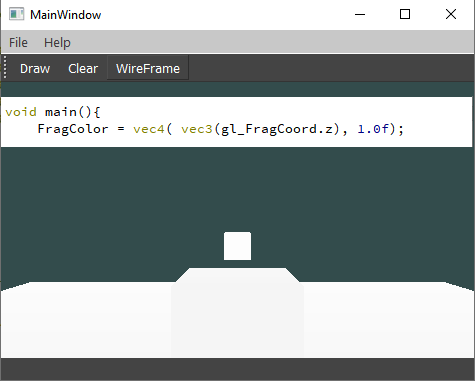


* + - * Trong fragment shader có 1 biến có sẵn là gl\_FragCoord(vec3) trong đó x và y là toạ độ vị trí của fragment trên màn hình và z chính là giá trị của đệm sâu cần so sánh.Chú ý cần phải kích hoạt depth test thì giá trị của z sẽ được cập nhật giá trị, nếu không kích hoạt depth test thì giá trị này sẽ không được ghi vào.

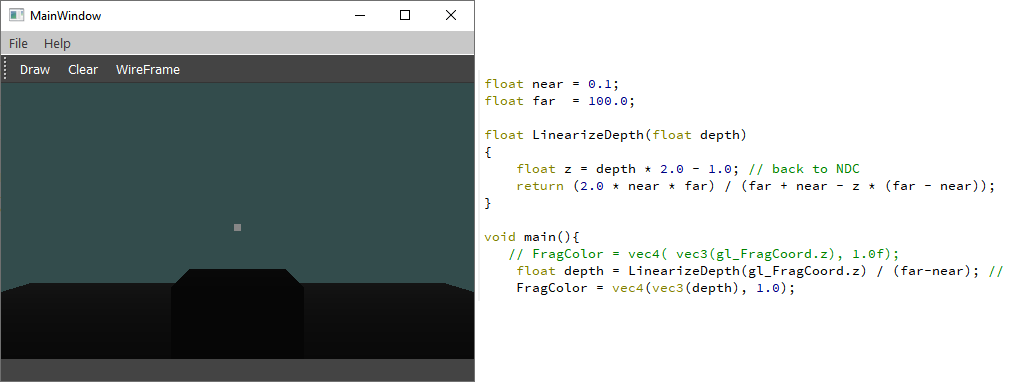
|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Type** | **Biến** | **Mô tả** |
| 1 | Fragment shader | vec4 gl\_FragCoord  min(0,0)  max(1200,900) | Tọa độ tương đối của current fragment trên cửa số window    // file Testshader.frag  #version 430  out vec4 color;  void main(void)  {  if (gl\_FragCoord.y < 295) color = vec4(1.0, 0.0, 0.0, 1.0); else color = vec4(0.0, 0.0, 1.0, 1.0);  } |

* + - * Hay nói cách khác khi vẽ các đối tượng trong không gian ta cần kích hoạt depth test để các đối tượng sẽ giống thực tế:
        + Đối tượng nào nằm trước, nằm sau,.
        + Đối tượng nào nhìn thấy, đối tượng nào bị khuất,….
        + OpenGL sẽ tự tính điều đó và hiển thị lên màn hình. Người dùng không cần tính toán lại giá trị này để chọn fragment nào sẽ được hiển thị nữa.
    1. Cách tính và lấy ra giá trị bộ đệm sâu trong fragment
       - Trong fragment shader có 1 biến cho biết độ sâu của fragment đó là gl\_FragCoord. Khi chạy hiển thị giá trị này thì toàn bộ đối tượng sẽ được hiển thị đều như là màu trắng tức z= 1.0 cho dù vật ở gần hay xa đều chung 1 giá trị này.Đây là cách tính giá trị đệm sâu mặc định của nó. Cách tính này chính là tuyến tính.

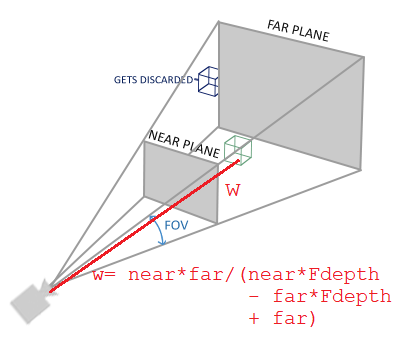




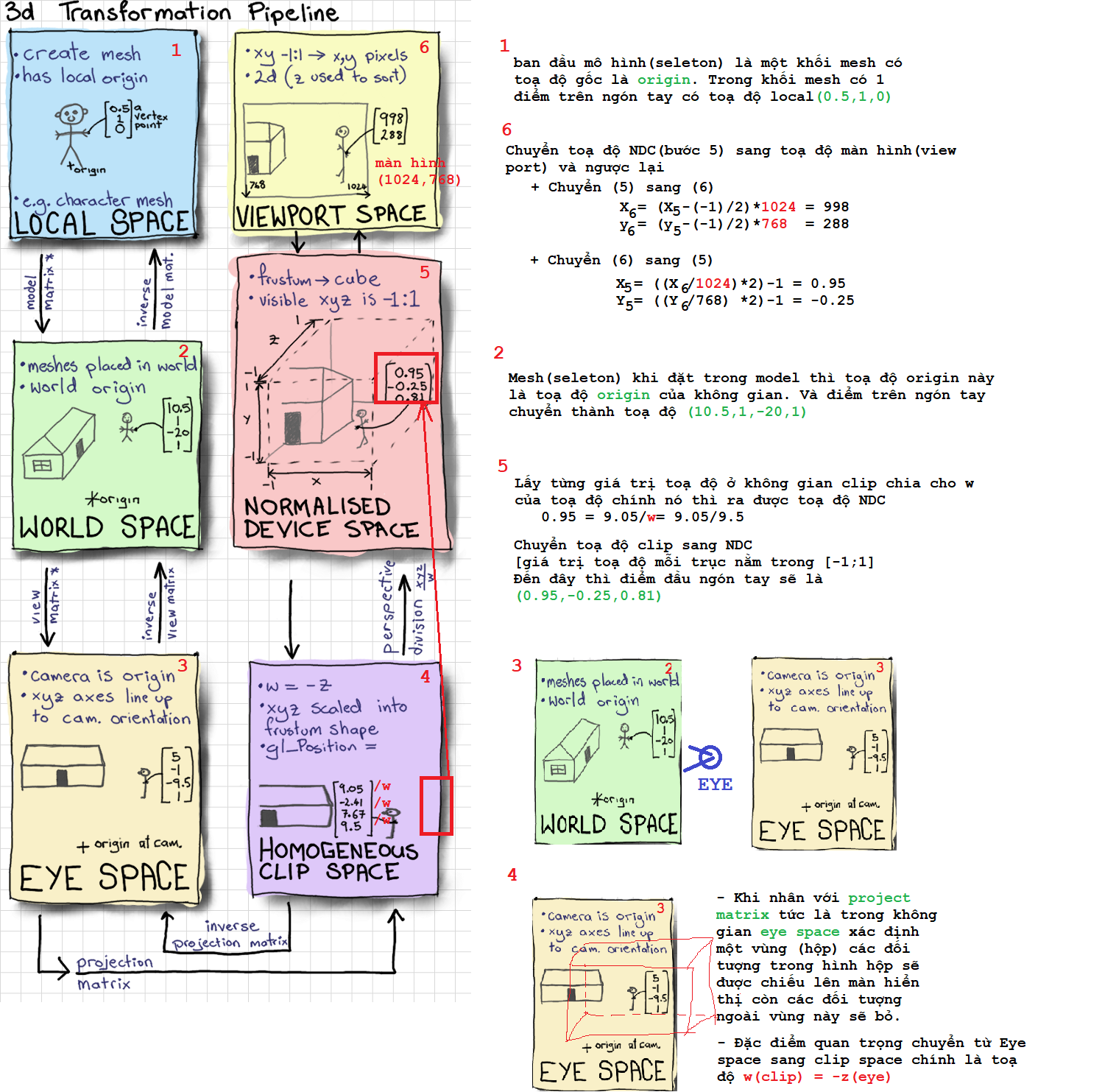
* + - * Theo cách tính trên thì các vật ở gần thì bộ đệm sâu sẽ gần như là như nhau. Như vậy họ đã phát triển ra một cách tính phi tuyến khác. Tức là các vật ở gần thì giá trị độ sâu càng khác nhau rõ rệt, càng ở xa thì giá trị này hầu như là như nhau. Do vậy khi tính theo phương pháp này độ sâu được tính chính xác hơn theo cách 1. Dưới đây sẽ tính lại depth value của từng frag dựa vào giá trị depth(linear) cảu biến gl\_FragCoord.



Từ đây ta có thể tính được khoảng cách từ fragment đến máy ảnh thông qua công thức sau:



1. Mouse picking với Ray casting
   1. Lý thuyết
      * + Link tham khảo: <https://antongerdelan.net/opengl/raycasting.html>
        + Toạ độ 3D trong các không gian.



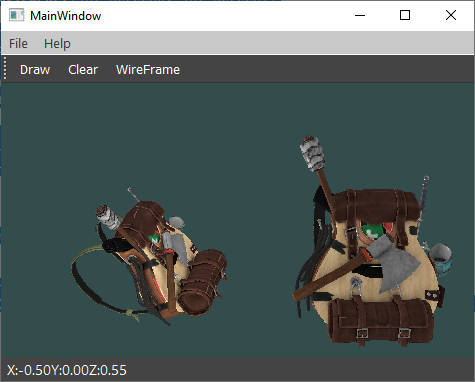
* + 1. Lấy toạ độ điểm click trên màn hình chuyển sang toạ độ global sử dụng glReadPixels

|  |
| --- |
| QVector4D AXBOpenGLWidget::**worldPosFromViewPort**(int screenX, int screenY)  {  float winz; // độ sâu của điểm trên viewport khi pick  // event->pos:toạ độ điểm lấy theo điểm gốc là (trên,trái)  // sử dụng glReadPixels thì toạ độ điểm x,y phải tính từ điểm gốc (dưới,trái).  int screenYTransform= this->height()- screenY;  glReadPixels(screenX, screenYTransform,1,1, GL\_DEPTH\_COMPONENT,GL\_FLOAT,&winz);  // Chuyển đổi toạ độ ViewSpace(màn hình) sang toạ độ NDC  float x= (2.0f\*screenX)/this->width()-1.0f;  float y = 1.0f-(2.0f\*screenY)/this->height();  float z= winz\*2.0f -1.0f;  //if(winz<1.0f){  float w= (2.0f \* \_near\*\_far)/(\_far+\_near-z\*(\_far - \_near));  QVector4D worldPos (x,y,z,1);  worldPos= w \* worldPos;  worldPos = view.inverted()\*projection.inverted()\*worldPos;  //qDebug() << "x: " << worldPos.x() << ",y: " << worldPos.y() << ",z: " <<worldPos.z() << ",w: " << w ;  //}  return worldPos;  } |

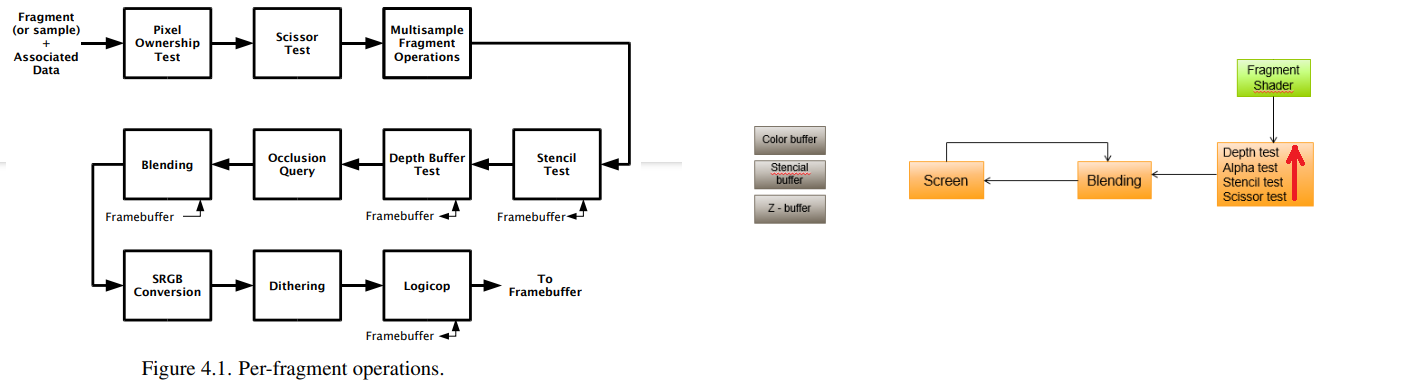
* 1. Thực hành

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include "camera.h"  #include <QPointF>  #include <QOpenGLShaderProgram>  #include <QTimer>  #include <QOpenGLTexture>  #include "mesh.h"  #include "model.h"  #include <QVector3D>  class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  struct **ModelInfo**{  Model \*model;  QVector3D worldPos;  float pitch;  float yaw;  float roll;  bool isSelected;  QString name;  };  explicit **AXBOpenGLWidget**(QWidget \*parent = nullptr);  ~***AXBOpenGLWidget***();  signals:  void **MousePickingPos**(QVector3D pos);  private slots:  void **on\_TimeOut**();  protected:  void ***initializeGL***();  void ***resizeGL***(int w, int h);  void ***paintGL***();  // Event  void ***mousePressEvent***(QMouseEvent \*event);  void ***mouseMoveEvent***(QMouseEvent \*event);  void ***wheelEvent***(QWheelEvent \*event);  void ***keyPressEvent***(QKeyEvent \*event);  QVector4D **worldPosFromViewPort**(int xOffset, int yOffset);  private:  Camera m\_camera;  QOpenGLShaderProgram shaderProgram, lightshaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model,modelLight;  QOpenGLTexture \*texDiffuse;  QOpenGLTexture \*texSpecular;  QMap<QString,ModelInfo> m\_Models;  bool isActiveEditModel;  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QMouseEvent>  #include <QWheelEvent>  #include <QKeyEvent>  #include <QDebug>  #include <QVector4D>  QPoint lastPos(0,0);  int perFrame = 100;// delta time / frame  float \_near = 0.1f;  float \_far = 100.0f;  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget(parent)  {  /\*  Cho phép kích hoạt sử dụng event keypress  nếu khong có dòng này mà vẫn có code  event key press thì nó cũng không chạy được  \*/  setFocusPolicy(Qt::StrongFocus);  /\* Cho phép kích hoạt sự kiện liên quan đến mouse\*/  //setMouseTracking(true);  m\_camera= Camera(QVector3D(0.0f, 1.0f, 3.0f));  m\_timer.setInterval(perFrame);  connect(&m\_timer, SIGNAL(timeout()), this, SLOT(on\_TimeOut()));  m\_timer.start();  // Ban đầu chưa select đối tượng thì chế độ active model sẽ là false  isActiveEditModel = false;  }  AXBOpenGLWidget::~***AXBOpenGLWidget***()  {  for(auto inter=m\_Models.begin(); inter!=m\_Models.end(); inter++){  ModelInfo \*modelInfo = &inter.value();  delete modelInfo->model;  }  }  void AXBOpenGLWidget::**on\_TimeOut**()  {  update();  }  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/light.vert");  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/light.frag");  lightshaderProgram.*link*();  success = lightshaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << lightshaderProgram.log();  // Texture  texDiffuse = new QOpenGLTexture(QImage(":/Images/Resources/Images/diffuseMapTex.png").mirrored());  texSpecular = new QOpenGLTexture(QImage(":/Images/Resources/Images/co.png").mirrored());  // Load model  auto m\_model= new Model(QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(),  "C:/Users/HUY NGUYEN/Desktop/OpenGL vs QT/OpenGLAndQT/Chuong 4\_Assimp/backpack/backpack.obj");  auto m\_model2= new Model(QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(),  "C:/Users/HUY NGUYEN/Desktop/OpenGL vs QT/OpenGLAndQT/Chuong 4\_Assimp/backpack/backpack.obj");  // đẩy models vào đối tượng m\_Models để quản lý dữ liệu  m\_Models["cube" + QString::number(1)]=ModelInfo{m\_model,QVector3D(0.0f,0.0f,0.0f),0.0f,0.0f,0.0f, false,QString::fromUtf8( "Hình hộp 1")};  m\_Models["cube" + QString::number(2)]=ModelInfo{m\_model2,QVector3D(5.0f,0.0f,6.0f),0.0f,0.0f,0.0f, false,QString::fromUtf8( "Hình hộp 2")};  }  void AXBOpenGLWidget::***resizeGL***(int w, int h)  {  Q\_UNUSED(w)  Q\_UNUSED(h)  }  QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  //QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  //QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.9f, 0.9f, 0.9f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(),\_near, \_far);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  model.setToIdentity();  //model.rotate(10, QVector3D(1.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  // draw models  foreach (auto modelInfo, m\_Models) {  model.setToIdentity();  model.translate(modelInfo.worldPos);  model.rotate(modelInfo.pitch, QVector3D(1.0f,0.0f,0.0f));  model.rotate(modelInfo.yaw, QVector3D(0.0f,1.0f,0.0f));  model.rotate(modelInfo.roll, QVector3D(0.0f,0.0f,1.0f));  shaderProgram.setUniformValue("model", model);  modelInfo.model->Draw(*shaderProgram*);  }  }  void AXBOpenGLWidget::***mousePressEvent***(QMouseEvent \*event)  {  makeCurrent();  lastPos = event->pos();  if(isActiveEditModel==true) return;  if(event->buttons() && Qt::LeftButton){  int screenX= event->pos().x();  // event->pos:toạ độ điểm lấy theo điểm gốc là (trên,trái)  // sử dụng glReadPixels thì toạ độ điểm x,y phải tính từ điểm gốc (dưới,trái).  int screenY= event->pos().y();  auto worldPosition = worldPosFromViewPort(screenX,screenY);  MousePickingPos(QVector3D(worldPosition));  // Selection  for(QMap<QString, ModelInfo>::iterator iter = m\_Models.begin(); iter!=m\_Models.end();iter++){  ModelInfo \*modelInfo = &iter.value();  // Tính bán kính của model  float r = (modelInfo->model->m\_maxY - modelInfo->model->m\_minY)/2;  // Tính toán khoảng cách từ điểm click đến điểm position đặt model  float lenthpick= modelInfo->worldPos.distanceToPoint(QVector3D(worldPosition));  if(lenthpick<r){  modelInfo->isSelected= true;  qDebug() << modelInfo->name <<lenthpick <<"<" << r<< "=" << modelInfo->isSelected << "WorldPos: " <<  modelInfo->worldPos.x() << ","<< modelInfo->worldPos.y() << "," << modelInfo->worldPos.z();  }else{  modelInfo->isSelected= false;  }  }  }  doneCurrent();  }  void AXBOpenGLWidget::***mouseMoveEvent***(QMouseEvent \*event)  {  makeCurrent();  auto posIn= event->pos();  // reversed since y-coordinates go from bottom to top  int xoffset = posIn.x() - lastPos.x();  int yoffset = lastPos.y() - posIn.y();  if(isActiveEditModel){  // move đối tượng  for (auto iter = m\_Models.begin(); iter != m\_Models.end();iter++) {  ModelInfo \*modelInfo = &iter.value();  if(modelInfo->isSelected==false) continue;  modelInfo->worldPos = QVector3D(worldPosFromViewPort(event->pos().x(), event->pos().y()));  }  }else {  if(event->buttons()& Qt::RightButton  || event->buttons()& Qt::LeftButton  || event->buttons()& Qt::MiddleButton )  if(event->buttons()& Qt::RightButton){  m\_camera.ProcessMouseMovement(xoffset, yoffset);  }else{  QPoint deltaPos= posIn - lastPos;  for (auto iter = m\_Models.begin(); iter != m\_Models.end();iter++) {  ModelInfo \*modelInfo = &iter.value();  if(modelInfo->isSelected==false) continue;  if( event->buttons()& Qt::MiddleButton){  modelInfo->roll += deltaPos.x();  }  if( event->buttons()& Qt::LeftButton){  modelInfo->yaw += deltaPos.x();  modelInfo->pitch += deltaPos.y();  }  }  }  }  lastPos=posIn;  update();  doneCurrent();  }  void AXBOpenGLWidget::***wheelEvent***(QWheelEvent \*event)  {  m\_camera.ProcessMouseScroll( event->angleDelta().y()/120);  update();  }  void AXBOpenGLWidget::***keyPressEvent***(QKeyEvent \*event)  {  float detatime2= 2.5\*perFrame/1000;  switch (event->key()) {  case Qt::Key\_W:m\_camera.ProcessKeyboard(FORWARD,detatime2); break;  case Qt::Key\_S:m\_camera.ProcessKeyboard(BACKWARD,detatime2); break;  case Qt::Key\_D:m\_camera.ProcessKeyboard(RIGHT,detatime2); break;  case Qt::Key\_A:m\_camera.ProcessKeyboard(LEFT,detatime2); break;  case Qt::Key\_M:isActiveEditModel=!isActiveEditModel; break;  }  update();  }  QVector4D AXBOpenGLWidget::**worldPosFromViewPort**(int screenX, int screenY)  {  float winz; // độ sâu của điểm trên viewport khi pick  // event->pos:toạ độ điểm lấy theo điểm gốc là (trên,trái)  // sử dụng glReadPixels thì toạ độ điểm x,y phải tính từ điểm gốc (dưới,trái).  int screenYTransform= this->height()- screenY;  glReadPixels(screenX, screenYTransform,1,1, GL\_DEPTH\_COMPONENT,GL\_FLOAT,&winz);  // Chuyển đổi toạ độ ViewSpace(màn hình) sang toạ độ NDC  float x= (2.0f\*screenX)/this->width()-1.0f;  float y = 1.0f-(2.0f\*screenY)/this->height();  float z= winz\*2.0f -1.0f;  //if(winz<1.0f){  float w= (2.0f \* \_near\*\_far)/(\_far+\_near-z\*(\_far - \_near));  QVector4D worldPos (x,y,z,1);  worldPos= w \* worldPos;  worldPos = view.inverted()\*projection.inverted()\*worldPos;  //qDebug() << "x: " << worldPos.x() << ",y: " << worldPos.y() << ",z: " <<worldPos.z() << ",w: " << w ;  //}  return worldPos;  } |

Kết quả di chuyển, quay đối tượng



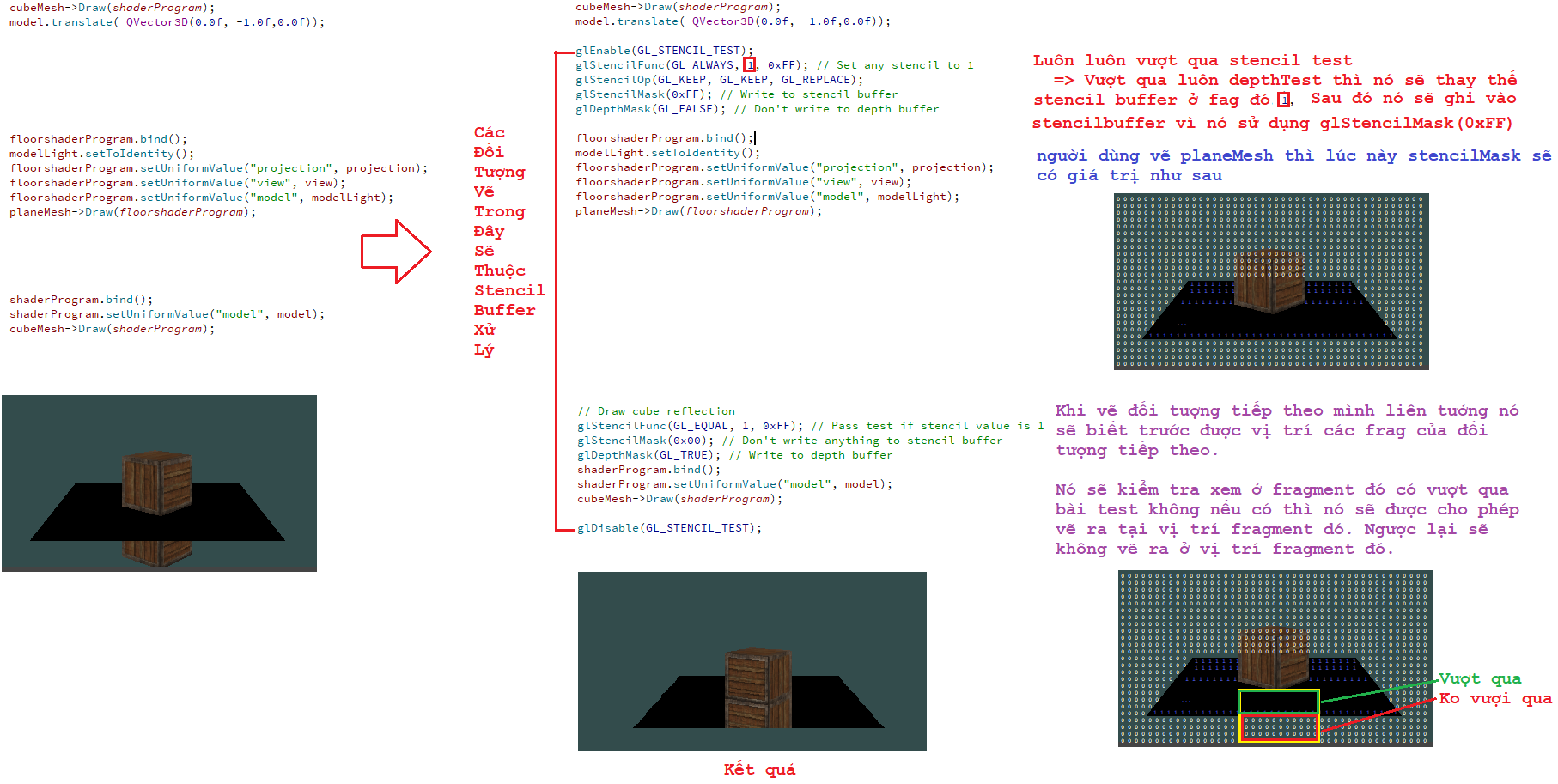
1. Các loại buffer và Test
   1. Danh sách các loại Test và buffer



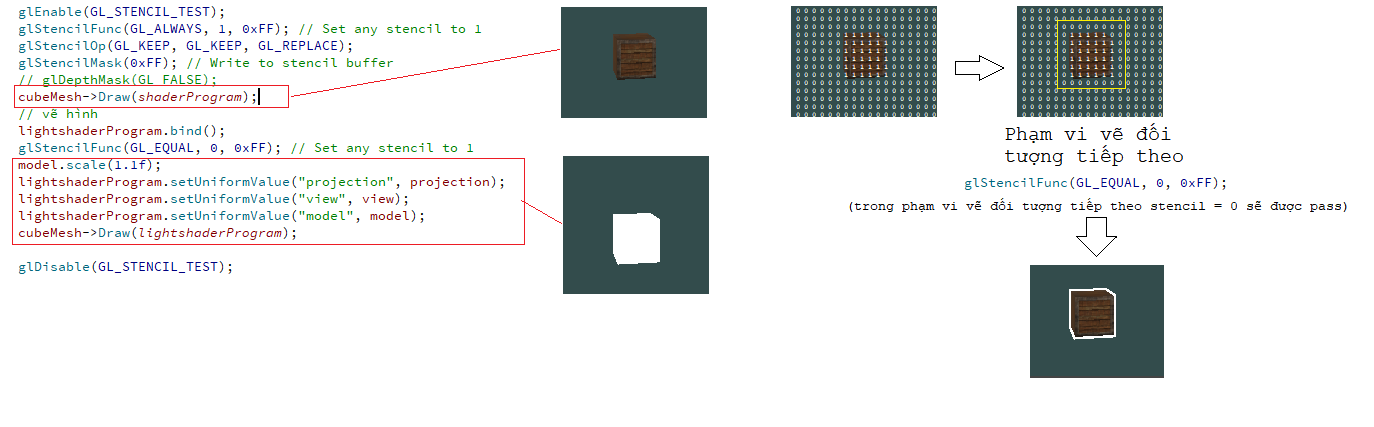
* 1. Thông tin của 1 fragment trước khi render hiển thị ra màn hình gồm có các dữ liệu sau
     + - Color buffer: Chứa màu cuối cùng khi hiển thị
       - Depth buffer: Chứa độ sâu(độ gần xa) của fragment, dữ liệu: 2- 4 byte.
       - Stencil buffer: Chứa giá trị stencil mặc định ban đầu sẽ là 0. Dữ liệu:1byte(8bit – chứa giá trị từ 0 đến 255)
  2. Depth buffer
     1. Lý thuyết
        + Mỗi một fragment sẽ chứa từ 2 đến 4 byte dữ liệu.
        + Bật tắt Depth mask
          - glDepthMask(GL\_FLASE): Khi sử dụng đoạn code này thì khi vẽ ở đằng sau nó vẫn đọc được bộ đệm sâu(quyết định cái nào vẽ trước cái nào vẽ sau) nhưng sau khi vẽ nó sẽ không cập nhật vào Depth buffer.
          - glDepthMask(GL\_TRUE): Khi vẽ xong đối tượng nó sẽ cập nhật vào depth buffer. Chú ý khi sử dụng đoạn code này chỉ áp dụng trước khi vẽ các đối tượng thông thường. Ngược lại khi vẽ các đối tượng còn lại(khác thường: có tính phản chiếu, blend,…) thì ta bắt buộc phải sử dụng đoạn code: glDepthMask(GL\_FLASE).(Thường trong khi sử dụng **stencil buffer , blend** ta bắt buộc phải sử dụng bật tắt Depth mask này)
     2. Thực hành
  3. Stencil buffer
     1. Lý thuyết
        + Mỗi một fragment(stencil) sẽ chứa 1byte dữ liệu : 0-255(8bit)
        + Stencil buffer thường dùng để vẽ đối tượng(hoặc nhiều đối tượng) hay một hay nhiều phần của các đối tượng đó ( hay nói cách khác là stencil dùng để vẽ hiệu ứng của đối tượng: phản chiếu, ouline,…).
        + Khi vẽ hiệu ứng của stencil thì các đối tượng vẽ nó được nằm trong đoạn code sau

|  |
| --- |
| glClear(..|GL\_STENCIL\_BUFFER\_BIT);  glEnable(GL\_STENCIL\_TEST);  // code  glDisable(GL\_STENCIL\_TEST); |

* + - * Trước khi vẽ đối tượng ta cần kích hoat glStencilFun(glStencilMask,glStencilOp,[có thể có thêm glDepthMask]) ít nhất phải sử dụng glStencilfun để xác định có vượt qua bài test không.
      * Quy trình sử dụng stencil buffer sẽ như sau
        + Ví dụ 1: Reflect sử dụng stencil <https://open.gl/depthstencils>



* + - * + Ví dụ 2: Outline sử dụng stencil



* + 1. Các phương thức stencil cần chú ý
       - * glStencilMask (Glunit mask)

nếu mask: 0x00 nó sẽ không cho phép thay đổi giá trị trong stencil buffer.

nếu mask: 0xFF: sau khi sử dụng glStencilFunc và glStencilOp thì dựa vào thông số trả về nó sẽ được ghi vào stencilbuffer.

* + - * + glStencilFunc: Đây là định nghĩa phương thức khi nào stencil pass khi nào không pass.

Cấu trúc: void glStencilFunc(Glenum func, Glint ref, Gluint mask)

glEnum func

|  |  |  |
| --- | --- | --- |
| **STT** | **func** | **Des** |
| 1 | GL\_NEVER | Always fails |
| 2 | GL\_LESS | Pass if(ref & mask) < (stencil &mask) |
| 3 | GL\_LEQUAL | Pass if(ref & mask) <= (stencil &mask) |
| 4 | GL\_GREATER | Pass if(ref & mask) > (stencil &mask) |
| 5 | GL\_GEQUAL | Pass if(ref & mask) >= (stencil &mask) |
| 6 | GL\_EQUAL | Pass if(ref & mask) = (stencil &mask) |
| 7 | GL\_NOTEQUAL | Pass if(ref & mask) != (stencil &mask) |
| 8 | GL\_ALWAYS | Always pass |

Ví dụ tính toán

glStencilFunc(ALWAYS,1,0): Luôn luôn vượt qua test stencil.

glStencilFunc(GL\_GREATER,1,0x00)

Bước 1: 1(0000 0001) AND 0x00=0(0000 0000) = 0(0000 0000)

Bước 2: Lấy kết quả của bước 1 AND với giá trị trong stencilbuffer. Với bất kỳ giá trị nào strong stencilbuffer khi AND 0 =0

Bước 3: Lấy kết quả bước 2 so sánh với giá trị ở bước 1 ở đây 0(bước 3) > 0 => phép kiểm tra stencil bị sai.

glStencilFunc(GL\_EQUAL,255,8)

Bước 1: 255(1111 1111) AND 8(0000 1000)= 8(0000 1000)

Bước 2: Giả sử giá trị trong stencil buffer là 1.Lấy kết quả bước 2: 8(0000 1000) AND 1(0000 0001) = 0 (0000 0000)

Bước 3: Lấy kết quả ở bước 2: so sánh với kết quả bước 1: 0(0000 0000) = 8(0000 1000) => phép kiểm tra stencil bị sai.

* + - * + glStencilOp:Ra lệnh xử lý các trường hợp sau khi test stencil.Dựa vào kết quả của test stencil.

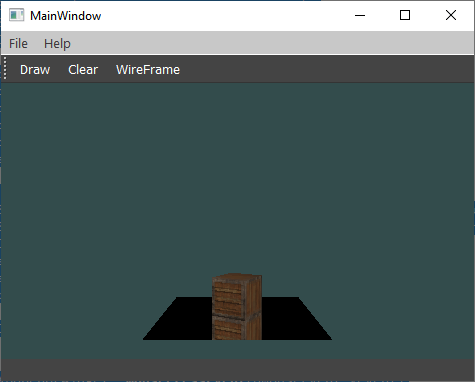
|  |  |
| --- | --- |
| GL\_KEEP | Giá trị stencil hiện được lưu trữ được giữ lại. |
| GL\_ZERO | Giá trị stencil được đặt thành 0. |
| GL\_REPLACE | Giá trị stencil được thay thế bằng giá trị tham chiếu(ref) được đặt bằng glStencilFunc. |
| GL\_INCR | Giá trị stencil được tăng lên 1nếu nó thấp hơn giá trị lớn nhất. |
| GL\_INCR\_WRAP | Tương tự như GL\_INCR , nhưng kết thúc nó trở lại 0 ngay sau khi giá trị lớn nhất bị vượt quá |
| GL\_DECR | Giá trị stencil sẽ giảm 1nếu nó cao hơn giá trị nhỏ nhất. |
| GL\_DECR\_WRAP | Tương tự như GL\_DECR , nhưng gói nó thành giá trị lớn nhất nếu nó kết thúc thấp hơn 0. |
| GL\_INVERT | Bitwise đảo ngược giá trị bộ đệm stencil hiện tại. |

Một số đoạn code hay sử dụng

* + 1. Thực hành
       - Tạo Reflect sử dụng stencil

|  |
| --- |
| #ifndef AXBOPENGLWIDGET\_H  #define AXBOPENGLWIDGET\_H  #include <QOpenGLWidget>  #include <QOpenGLFunctions\_3\_3\_Core>  #include "camera.h"  #include <QPointF>  #include <QOpenGLShaderProgram>  #include <QTimer>  #include <QOpenGLTexture>  #include "mesh.h"  class AXBOpenGLWidget : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  Q\_OBJECT  public:  explicit AXBOpenGLWidget(QWidget \*parent = nullptr);  private slots:  void on\_TimeOut();  protected:  void initializeGL();  void resizeGL(int w, int h);  void paintGL();  // Event  void mousePressEvent(QMouseEvent \*event);  void mouseMoveEvent(QMouseEvent \*event);  void wheelEvent(QWheelEvent \*event);  void keyPressEvent(QKeyEvent \*event);  private:  Camera m\_camera;  QOpenGLShaderProgram shaderProgram, lightshaderProgram, floorshaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model,modelLight;  QOpenGLTexture \*texDiffuse;  QOpenGLTexture \*texSpecular;  Mesh \*cubeMesh;  Mesh \*planeMesh;  Mesh \* processMesh(float \*vertices, int size , unsigned int textureId);  };  #endif // AXBOPENGLWIDGET\_H |
| #include "axbopenglwidget.h"  #include <QMouseEvent>  #include <QWheelEvent>  #include <QKeyEvent>  #include <QDebug>  QPoint lastPos(0,0);  int perFrame = 100;// delta time / frame  float \_near = 0.1f;  float \_far = 100.0f;  AXBOpenGLWidget::**AXBOpenGLWidget**(QWidget \*parent) : QOpenGLWidget(parent)  {  ...  }  void AXBOpenGLWidget::**on\_TimeOut**()  {  update();  }  float cubeVertices[] = {  // positions // texture Coords  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  ...  };  float planeVertices[] = {  // positions // texture Coords (note we set these higher than 1 (together with GL\_REPEAT as texture wrapping mode). this will cause the floor texture to repeat)  2.0f, -0.5f, 2.0f, 2.0f, 0.0f,  -2.0f, -0.5f, 2.0f, 0.0f, 0.0f,  -2.0f, -0.5f, -2.0f, 0.0f, 2.0f,  2.0f, -0.5f, 2.0f, 2.0f, 0.0f,  -2.0f, -0.5f, -2.0f, 0.0f, 2.0f,  2.0f, -0.5f, -2.0f, 2.0f, 2.0f  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/light.vert");  lightshaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/light.frag");  lightshaderProgram.*link*();  success = lightshaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << lightshaderProgram.log();  floorshaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/floor.vert");  floorshaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/floor.frag");  floorshaderProgram.*link*();  success = floorshaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << floorshaderProgram.log();  // Texture  texDiffuse = new QOpenGLTexture(QImage(":/Images/Resources/Images/diffuseMapTex.png").mirrored());  texSpecular = new QOpenGLTexture(QImage(":/Images/Resources/Images/co.png").mirrored());  cubeMesh= processMesh(cubeVertices,36,texDiffuse->textureId());  planeMesh= processMesh(planeVertices,6, texSpecular->textureId());  model.setToIdentity();  model.translate( QVector3D(0.0f, -1.0f,0.0f));  }  QVector3D material\_ambient= QVector3D( 1.0f, 0.5f, 0.31f);  //QVector3D material\_diffuse=QVector3D(1.0f, 0.5f, 0.31f);  //QVector3D material\_specular=QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_ambient= QVector3D(0.9f, 0.9f, 0.9f);  QVector3D light\_diffuse= QVector3D(0.5f, 0.5f, 0.5f);  QVector3D light\_specular= QVector3D(1.0f, 1.0f, 1.0f);  QVector3D light\_color = QVector3D(1.0f, 1.0f,0.0f);  QVector3D light\_pos= QVector3D(0.0f, -1.0f, 5.0f);  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_COLOR\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  //model.setToIdentity();  model.translate( QVector3D(0.0f, 1.0f,0.0f));  model.rotate(10, QVector3D(0.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  cubeMesh->Draw(*shaderProgram*);  model.translate( QVector3D(0.0f, -1.0f,0.0f));  glEnable(GL\_STENCIL\_TEST);  glStencilFunc(GL\_ALWAYS, 1, 0xFF); // Set any stencil to 1  glStencilOp(GL\_KEEP, GL\_KEEP, GL\_REPLACE);  glStencilMask(0xFF); // Write to stencil buffer  glDepthMask(GL\_FALSE); // Don't write to depth buffer  floorshaderProgram.bind();  modelLight.setToIdentity();  floorshaderProgram.setUniformValue("projection", projection);  floorshaderProgram.setUniformValue("view", view);  floorshaderProgram.setUniformValue("model", modelLight);  planeMesh->Draw(*floorshaderProgram*);  // Draw cube reflection  glStencilFunc(GL\_EQUAL, 1, 0xFF); // Pass test if stencil value is 1  glStencilMask(0x00); // Don't write anything to stencil buffer  glDepthMask(GL\_TRUE); // Write to depth buffer  shaderProgram.bind();  shaderProgram.setUniformValue("model", model);  cubeMesh->Draw(*shaderProgram*);  glDisable(GL\_STENCIL\_TEST);  }  Mesh \*AXBOpenGLWidget::**processMesh**(float \*vertices, int size, unsigned int textureId)  {  vector<Vertex> \_vertices;  vector<unsigned int> \_indices;  vector<Texture> \_textures;  for (int i=0; i<size; i++) {  Vertex vert;  vert.Position[0]=vertices[i\*5+0];  vert.Position[1]=vertices[i\*5+1];  vert.Position[2]=vertices[i\*5+2];  vert.TexCoords[0]=vertices[i\*5+3];  vert.TexCoords[1]=vertices[i\*5+4];  \_vertices.push\_back(vert);  \_indices.push\_back(i);  }  Texture tex;  tex.id= textureId;  tex.type ="texture\_specular";  \_textures.push\_back(tex);  return new Mesh(QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(), \_vertices, \_indices,\_textures);  } |

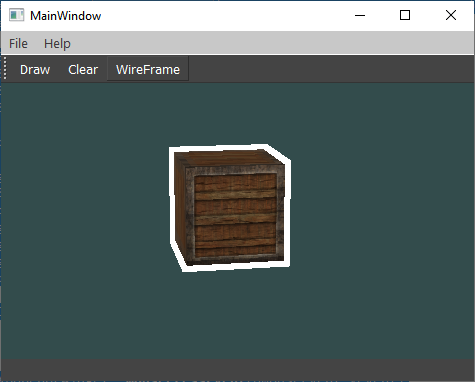
Kết quả



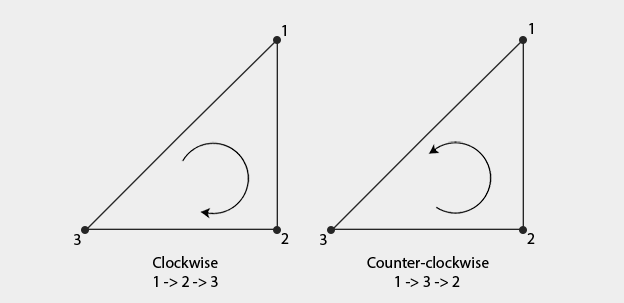
* + - * Tạo outline sử dụng stencil

|  |
| --- |
| class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  ..  Mesh \*cubeMesh;  Mesh \*planeMesh;  Mesh \* **processMesh**(float \*vertices, int size , unsigned int textureId);  };  #endif // AXBOPENGLWIDGET\_H |
| void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_COLOR\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  model.setToIdentity();  model.rotate(10, QVector3D(0.0f, 1.0f, 0.0f));  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", material\_ambient);  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",light\_ambient);  shaderProgram.setUniformValue("light.diffuse", light\_diffuse);  shaderProgram.setUniformValue("light.specular", light\_specular);  shaderProgram.setUniformValue("light.lightColor", light\_color);  shaderProgram.setUniformValue("light.lightPos", light\_pos);  glEnable(GL\_STENCIL\_TEST);  glStencilFunc(GL\_ALWAYS, 1, 0xFF); // Set any stencil to 1  glStencilOp(GL\_KEEP, GL\_KEEP, GL\_REPLACE);  glStencilMask(0xFF); // Write to stencil buffer  // glDepthMask(GL\_FALSE);  cubeMesh->Draw(*shaderProgram*);  // vẽ hình  lightshaderProgram.bind();  glStencilFunc(GL\_EQUAL, 0, 0xFF); // Set any stencil to 1  model.scale(1.1f);  lightshaderProgram.setUniformValue("projection", projection);  lightshaderProgram.setUniformValue("view", view);  lightshaderProgram.setUniformValue("model", model);  cubeMesh->Draw(*lightshaderProgram*);  glDisable(GL\_STENCIL\_TEST);  } |

Kết quả



1. Blend
2. Face culing
   1. Lý thuyết
      * + Khi ta vẽ một đối tượng cube thì tối đa ta có thể nhìn là 3 mặt. Mặc định khi vẽ trong opengl thì nó sẽ vẽ cả 6 mặt. Như vậy nó sẽ hiển thị những phần bị ẩn không cần thiết.
        + Để tăng tốc lên tốc độ 50% khi vẽ thì ta sẽ không hiển thị những mặt ẩn lên. Trong opengl sẽ có tính năng là Face Culing
        + Khi bật cull face thì mặc định nó chỉ hiển thị những mặt Front(mặt trước – toạ độ điểm theo chiều ngược chiều kim đồng hồ)



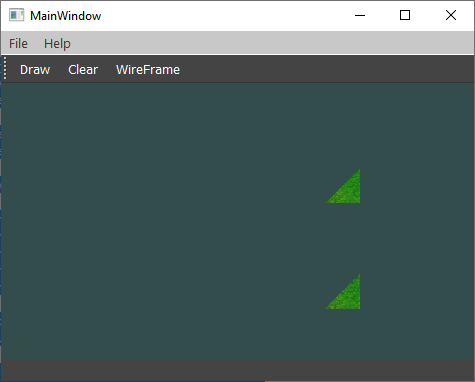
* + - * Các phương thức cần chú ý:

|  |
| --- |
| glEnable(GL\_CULL\_FACE); // Bật chế độ cull face  // Thiết lập mặt trước là CW(Theo chiều kim đồng hồ) CCW(Ngược chiều kim đồng hồ)  glFrontFace(GL\_CW);  // mặc định là GL\_BACK: Không hiển thị mặt sau,chỉ hiển thị mặt trước.  // GL\_FRONT : Không hiển thị mặt trước, Chỉ hiển thị mặt sau.  // GL\_FRONT\_AND\_BACK : Không hiển thị mặt trước lẫn mặt sau.  glCullFace(GL\_BACK); |

* 1. Thực hành

|  |
| --- |
| class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  ...  Mesh \*NKDHMesh;  Mesh \*CKDHMesh;  Mesh \* **processMesh**(float \*vertices, int size , unsigned int textureId);  }; |
| // Ngược chiều kim đồng hồ  float Vertices\_NguocKDH[] = {  // positions // texture Coords  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  0.5f, -0.5f, -0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  };  // Cùng chiều kim đồng hồ  float CCLVertices\_CungKDH[] = {  // positions // texture Coords  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, -0.5f, 1.0f, 0.0f,  };  void AXBOpenGLWidget::***initializeGL***()  {  *...*  NKDHMesh= processMesh(Vertices\_NguocKDH,3,texDiffuse->textureId());  CKDHMesh= processMesh(CCLVertices\_CungKDH,3,texSpecular->textureId());  model.setToIdentity();  }  void AXBOpenGLWidget::***paintGL***()  {  ...  glEnable(GL\_CULL\_FACE); // Bật chế độ cull face  // Thiết lập mặt trước là CW(Theo chiều kim đồng hồ)  // CCW(Ngược chiều kim đồng hồ)  glFrontFace(GL\_CW);  // mặc định là GL\_BACK: Không hiển thị mặt sau,chỉ hiển thị mặt trước.  // GL\_FRONT : Không hiển thị mặt trước, Chỉ hiển thị mặt sau.  // GL\_FRONT\_AND\_BACK : Không hiển thị mặt trước lẫn mặt sau.  glCullFace(GL\_BACK);    // Cùng chiều kim đồng hồ  NKDHMesh->Draw(*shaderProgram*);  model.translate(0.0f,3.0f,0.0f);  shaderProgram.setUniformValue("model", model);  NKDHMesh->Draw(*shaderProgram*);  // Ngược chiều kim đồng hồ  model.translate(3.0f,-3.0f,0.0f);  shaderProgram.setUniformValue("model", model);  CKDHMesh->Draw(*shaderProgram*);  model.translate(0.0f,3.0f,0.0f);  shaderProgram.setUniformValue("model", model);  CKDHMesh->Draw(*shaderProgram*);  glDisable(GL\_CULL\_FACE);  } |

Kết quả



1. Framebuffer và renderbuffer
   1. FrameBuffer
      1. Lý thuyết
         * Sử dụng : Khi ảnh 3D được render ra màn hình, Làm sao để ta có thể lấy được dữ liệu đó để xử lý hậu kỳ rồi mới hiển thị 1 phần hay toàn bộ trên màn hình ở 1 vị trí bất kỳ thì framebuffer sẽ giải quyết được vấn đề đó.Ví dụ: framebuffer sẽ tạo 1 vùng trên màn hình để hiển thị toàn bộ dữ liệu có trên màn hình(có/không xử lý hậu kỳ)
         * FrameBuffer là một tập hợp các bộ đệm có thể được sử dụng làm đích để kết xuất. OpenGL có hai loại bộ đệm khung: Bộ đệm khung mặc định , được cung cấp bởi OpenGL Context ; và bộ đệm khung do người dùng tạo được Framebuffer Objects (FBOs).
         * Một FBO có một hoặc nhiều attachments(đính kèm)
           + GL\_COLOR\_ATTACHMENT
           + GL\_DEPTH\_ATTACHMENT
           + GL\_STENCIL\_ATTACHMENT
           + GL\_DEPTH\_STENCIL\_ATTACHMENT(depth and stencil interleaved into one)
         * Mỗi phần đính kèm là một Texture hoặc một Renderbuffer Object.
         * không giống như Textures, renderbuffers have no mip-máp, no layers or tiling , nor texture parameters , and they cannot be sampled.
         * FBO mặc định có 4 màu đính kèm: FRONT\_LEFT, FRONT\_RIGHT, BACK\_LEFT, BACK\_RIGHT
      2. Attaching Images
         * Các phương pháp gán texture vào framebuffer

|  |
| --- |
| void glFramebufferTexture1D(GLenum target​, GLenum attachment​, GLenum textarget​, GLuint texture​, GLint level​);  void glFramebufferTexture2D(GLenum target​, GLenum attachment​, GLenum textarget​, GLuint texture​, GLint level​);  void glFramebufferTextureLayer(GLenum target​, GLenum attachment​, GLuint texture​, GLint level​, GLint layer​); |

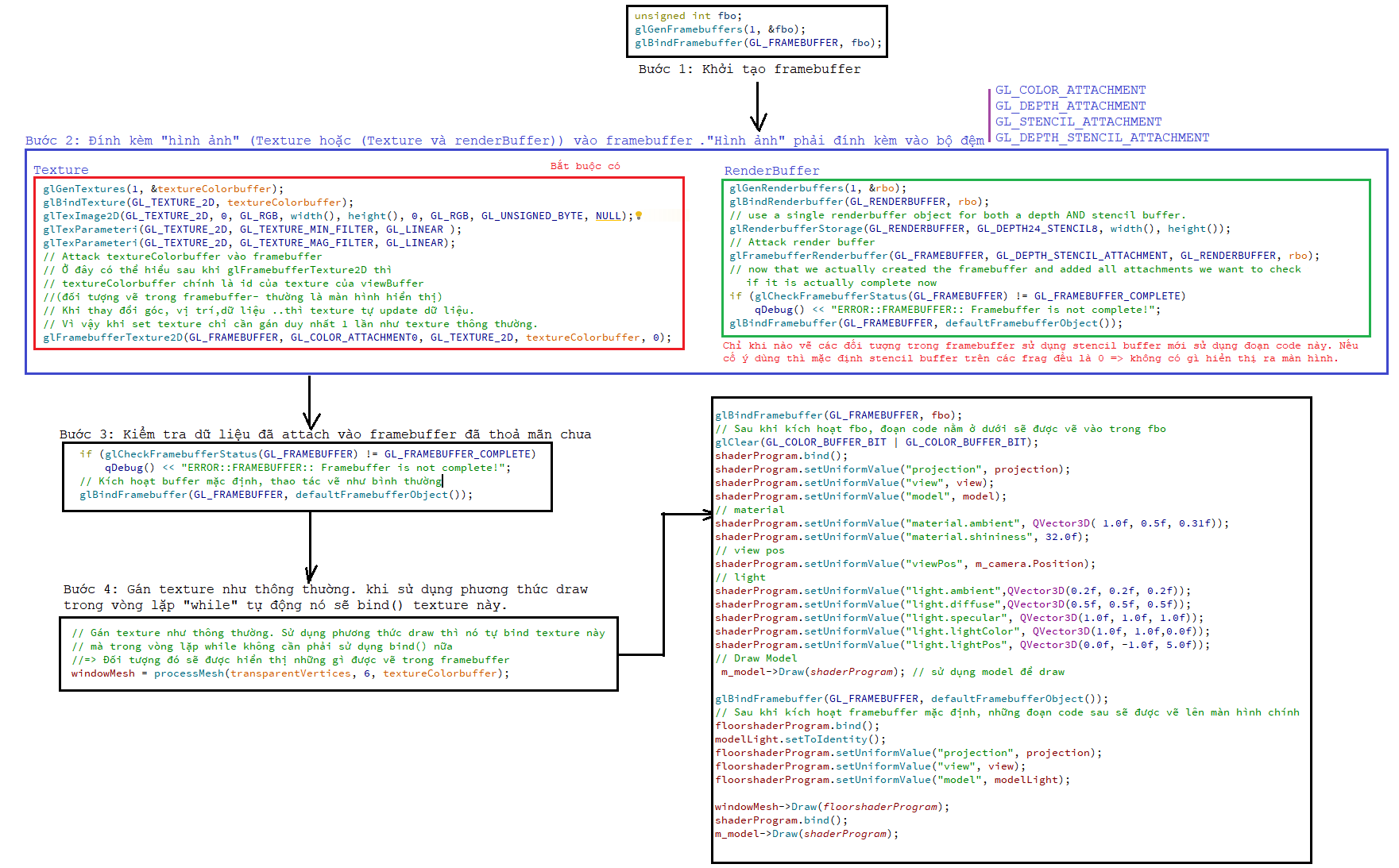
Trong đó

|  |  |  |
| --- | --- | --- |
| **STT** | **Tham số** | **Mô tả** |
| 1 | GLenum target | Loại frameBuffer  + GL\_FRAMEBUFFER: đọc và ghi vào buffer  + GL\_READ\_FRAMEBUFFER: chỉ đọc  + GL\_DRAW\_FRAMEBUFFER: chỉ ghi |
| 2 | Glenum attachment | Loại tệp đính kèm mà chúng ta muốn đính kèm.  + GL\_COLOR\_ATTACHMENT  + GL\_DEPTH\_ATTACHMENT  + GL\_STENCIL\_ATTACHMENT  + GL\_DEPTH\_STENCIL\_ATTACHMENT |
| 3 | GLenum textarget | Loại kết cấu muốn đính kèm như: GL\_TEXTURE\_1D, GL\_TEXTURE\_2D, GL\_TEXTURE\_2D\_MULTISAMPLE,… |
| 4 | GLuint texture | kết cấu thực tế muốn đính kèm |
| 5 | GLint level | mức mipmap |

* 1. Renderbuffer Object(rdo)
     1. Lý thuyết
        + Renderbuffer là bộ đệm thực tế (actual buffer) e.g. một mảng bytes, integers, pixels hoặc bất kỳ thứ gì. Tuy nhiên, a renderbuffer object không thể đọc trực tiếp từ chúng. nhưng có thể đọc từ chúng qua tốc độ chậmglReadPixels Điều này trả về một vùng pixel được chỉ định từ framebuffer hiện được liên kết, nhưng không trực tiếp từ chính tệp đính kèm.
        + Vì dữ liệu của họ ở định dạng gốc nên chúng khá nhanh khi ghi dữ liệu hoặc sao chép dữ liệu sang các bộ đệm khác. glfwSwapBuffers
     2. Nội dung

|  |
| --- |
| unsigned int rbo;  glGenRenderbuffers(1, &rbo);  glBindRenderbuffer(GL\_RENDERBUFFER, rbo);  // Tạo Depth và stencil buffer sử dụng glRenderbufferStorage đuôi chính là 24 và 8(bits)  glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH24\_STENCIL8, 800, 600);  // Đính kèm vào renderbuffer  glFramebufferRenderbuffer(GL\_FRAMEBUFFER, GL\_DEPTH\_STENCIL\_ATTACHMENT, GL\_RENDERBUFFER, rbo); |

* 1. Các bước xử lý chung



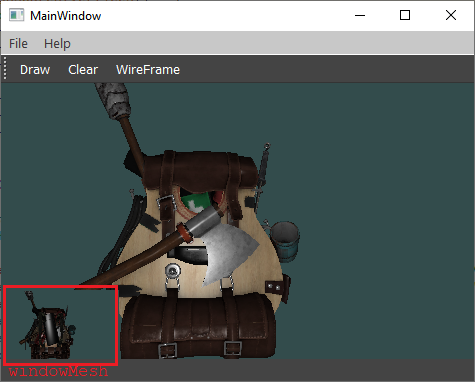
* 1. Post-processing(Xử lý hậu kỳ)

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Tên** | **Code(frag)** | **Kết quả** |
| 1 |  | FragColor = vec4(vec3(texture(screenTexture, TexCoords)), 1.0); |  |
| 2 | Inversion | FragColor = vec4(vec3(1.0 - texture(screenTexture, TexCoords)), 1.0); |  |
| 3 | Grayscale | FragColor = texture(screenTexture, TexCoords);  float average = (FragColor.r + FragColor.g + FragColor.b) / 3.0;  FragColor = vec4(average, average, average, 1.0); |  |
| 4 |  | FragColor = texture(screenTexture, TexCoords);  float average = 0.2126 \* FragColor.r + 0.7152 \* FragColor.g + 0.0722 \* FragColor.b;  FragColor = vec4(average, average, average, 1.0); |  |
| 5 | Kernel effects | const float offset = 1.0 / 300.0;  void main()  { vec2 offsets[9] = vec2[](  vec2(-offset, offset), // top-left  vec2( 0.0f, offset), // top-center  vec2( offset, offset), // top-right  vec2(-offset, 0.0f), // center-left  vec2( 0.0f, 0.0f), // center-center  vec2( offset, 0.0f), // center-right  vec2(-offset, -offset), // bottom-left  vec2( 0.0f, -offset), // bottom-center  vec2( offset, -offset) // bottom-right  );  float kernel[9] = float[](  -1, -1, -1,  -1, 9, -1,  -1, -1, -1  );    vec3 sampleTex[9];  for(int i = 0; i < 9; i++)  {  sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));  }  vec3 col = vec3(0.0);  for(int i = 0; i < 9; i++)  col += sampleTex[i] \* kernel[i];    FragColor = vec4(col, 1.0);    } | Blur:  float kernel[9] = float[](  1.0 / 16, 2.0 / 16, 1.0 / 16,  2.0 / 16, 4.0 / 16, 2.0 / 16,  1.0 / 16, 2.0 / 16, 1.0 / 16  );  Edge detection:  float kernel[9] = float[](  -1, -1, -1,  -1, -8, -1,  -1, -1, -1  ); |

* 1. Thực hành
     1. Ví dụ 1:

|  |
| --- |
| class **AXBOpenGLWidget** : public QOpenGLWidget,QOpenGLFunctions\_3\_3\_Core  {  ...  private:  Camera m\_camera;  QOpenGLShaderProgram shaderProgram, displayshaderProgram;  QTimer m\_timer;  QMatrix4x4 projection,view,model,modelLight;  Model \* m\_model;  unsigned int fbo;  Mesh \*windowMesh;  Mesh \***processMesh**(float \*vertices, int size, unsigned int textureId);  }; |
| float transparentVertices[] = {  // positions // texture Coords  -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,  -0.5f, -1.0f, 0.0f, 1.0f, 0.0f,  -1.0f, -0.5f, 0.0f, 0.0f, 1.0f,  -1.0f, -0.5f, 0.0f, 0.0f, 1.0f,  -0.5f,-0.5f, 0.0f, 1.0f, 1.0f,  -0.5f, -1.0f, 0.0f, 1.0f, 0.0f,  };  void AXBOpenGLWidget::***initializeGL***()  {  *initializeOpenGLFunctions*();  // shader  bool success;  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  displayshaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/fbuffer.vert");  displayshaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/fbuffer.frag");  displayshaderProgram.*link*();  success = displayshaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << displayshaderProgram.log();  //Bước 1: Khởi tạo  glGenFramebuffers(1, &fbo);  glBindFramebuffer(GL\_FRAMEBUFFER, fbo);  // generate texture  unsigned int textureColorbuffer;  glGenTextures(1, &textureColorbuffer);  glBindTexture(GL\_TEXTURE\_2D, textureColorbuffer);  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width(), height(), 0, GL\_RGB, GL\_UNSIGNED\_BYTE, NULL);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR );  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  // Attack textureColorbuffer vào framebuffer  // Ở đây có thể hiểu sau khi glFramebufferTexture2D thì  // textureColorbuffer chính là id của texture của viewBuffer  //(đối tượng vẽ trong framebuffer- thường là màn hình hiển thị)  // Khi thay đổi góc, vị trí,dữ liệu ..thì texture tự update dữ liệu.  // Vì vậy khi set texture chỉ cần gán duy nhất 1 lần như texture thông thường.  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, textureColorbuffer, 0);  if (glCheckFramebufferStatus(GL\_FRAMEBUFFER) != GL\_FRAMEBUFFER\_COMPLETE)  qDebug() << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!";  // Kích hoạt buffer mặc định, thao tác vẽ như bình thường  glBindFramebuffer(GL\_FRAMEBUFFER, defaultFramebufferObject());  // Sử dụng mesh  m\_model= new Model(QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions\_3\_3\_Core>(),  "C:/Users/HUY NGUYEN/Desktop/OpenGL vs QT/OpenGLAndQT/Chuong 4\_Assimp/backpack/backpack.obj");  // Gán texture như thông thường. Sử dụng phương thức draw thì nó tự bind texture này  // mà trong vòng lặp while không cần phải sử dụng bind() nữa  //=> Đối tượng đó sẽ được hiển thị những gì được vẽ trong framebuffer  windowMesh = processMesh(transparentVertices, 6, textureColorbuffer);  }  void AXBOpenGLWidget::***paintGL***()  {  glClearColor(0.2f,0.3f,0.3f,1.0f);  glEnable(GL\_DEPTH\_TEST);  glDepthFunc(GL\_LESS);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_COLOR\_BUFFER\_BIT);  projection.setToIdentity();  projection.perspective(m\_camera.Zoom, (float)width()/height(), 0.1f, 100.0f);  view.setToIdentity();  view= m\_camera.GetViewMatrix();  model.setToIdentity();  glBindFramebuffer(GL\_FRAMEBUFFER, fbo);  // Sau khi kích hoạt fbo, đoạn code nằm ở dưới sẽ được vẽ vào trong fbo  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_COLOR\_BUFFER\_BIT);  shaderProgram.bind();  shaderProgram.setUniformValue("projection", projection);  shaderProgram.setUniformValue("view", view);  shaderProgram.setUniformValue("model", model);  // material  shaderProgram.setUniformValue("material.ambient", QVector3D( 1.0f, 0.5f, 0.31f));  shaderProgram.setUniformValue("material.shininess", 32.0f);  // view pos  shaderProgram.setUniformValue("viewPos", m\_camera.Position);  // light  shaderProgram.setUniformValue("light.ambient",QVector3D(0.2f, 0.2f, 0.2f));  shaderProgram.setUniformValue("light.diffuse",QVector3D(0.5f, 0.5f, 0.5f));  shaderProgram.setUniformValue("light.specular", QVector3D(1.0f, 1.0f, 1.0f));  shaderProgram.setUniformValue("light.lightColor", QVector3D(1.0f, 1.0f,0.0f));  shaderProgram.setUniformValue("light.lightPos", QVector3D(0.0f, -1.0f, 5.0f));  // Draw Model  m\_model->Draw(*shaderProgram*); // sử dụng model để draw  glBindFramebuffer(GL\_FRAMEBUFFER, defaultFramebufferObject());  // Sau khi kích hoạt framebuffer mặc định, những đoạn code sau sẽ được vẽ lên màn hình chính  displayshaderProgram.bind();  modelLight.setToIdentity();  displayshaderProgram.setUniformValue("projection", projection);  displayshaderProgram.setUniformValue("view", view);  displayshaderProgram.setUniformValue("model", modelLight);  /\*  \* Vì windowMesh lúc trước người dùng đã gán texture(framebuffer) vào đối tượng này rồi  \* Nên các hiệu ứng hiển thị Post-processing sẽ được viết vào trong displayshaderProgram.  \*/  windowMesh->Draw(*displayshaderProgram*);  shaderProgram.bind();  m\_model->Draw(*shaderProgram*);  } | |
| // fbuffer.vert  #version 330 core  layout (location = 0) in vec3 aPos;  layout (location = 2) in vec2 aTexCoords;  out vec2 TexCoords;  uniform mat4 projection;  uniform mat4 view;  uniform mat4 model;  void main()  {  TexCoords = aTexCoords;  //nếu muốn vẽ trong không gian tính toán các thông số khác như thông thường:  //ví dụ:projection \* view\* model \* vec4(aPos.x,aPos.y,aPos.z,1.0);  // Ở đây muốn vẽ đối tượng ở góc 1/16 dưới cùng bên trái màn hình  gl\_Position = vec4(aPos.x, aPos.y,aPos.z, 1.0);  } | |
| // fbuffer.frag: chứa code vẽ các hiệu ứng khi hiển thị  #version 330 core  out vec4 FragColor;  in vec2 TexCoords;  uniform sampler2D screenTexture;  const float offset = 1.0 / 300.0;  void main()  {  //====================Hiệu ứng thông thường  FragColor = vec4(vec3(texture(screenTexture, TexCoords)), 1.0);  // //====================Hiệu ứng đảo ngược(Inversion)  // FragColor = vec4(vec3(1.0 - texture(screenTexture, TexCoords)), 1.0);  // //====================Hiệu ứng: Grayscale1  // FragColor = texture(screenTexture, TexCoords);  // float average = (FragColor.r + FragColor.g + FragColor.b) / 3.0;  // FragColor = vec4(average, average, average, 1.0);  // //====================Hiệu ứng: Grayscale2  // FragColor = texture(screenTexture, TexCoords);  // float average = 0.2126 \* FragColor.r + 0.7152 \* FragColor.g + 0.0722 \* FragColor.b;  // FragColor = vec4(average, average, average, 1.0);  // //====================Hiệu ứng: Kernel effects  // vec2 offsets[9] = vec2[](  // vec2(-offset, offset), // top-left  // vec2( 0.0f, offset), // top-center  // vec2( offset, offset), // top-right  // vec2(-offset, 0.0f), // center-left  // vec2( 0.0f, 0.0f), // center-center  // vec2( offset, 0.0f), // center-right  // vec2(-offset, -offset), // bottom-left  // vec2( 0.0f, -offset), // bottom-center  // vec2( offset, -offset) // bottom-right  // );  // // Chọn một trong các loại ma trận có các hiệu ứng khác nhau  // // 0. default  // float kernel[9] = float[](  // -1, -1, -1,  // -1, 9, -1,  // -1, -1, -1  // );  //// // 1. blur  //// float kernel[9] = float[](  //// 1.0 / 16, 2.0 / 16, 1.0 / 16,  //// 2.0 / 16, 4.0 / 16, 2.0 / 16,  //// 1.0 / 16, 2.0 / 16, 1.0 / 16  //// );  //// //2.Edge detection  //// float kernel[9] = float[](  //// -1, -1, -1,  //// -1, -8, -1,  //// -1, -1, -1  //// );  // vec3 sampleTex[9];  // for(int i = 0; i < 9; i++)  // {  // sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));  // }  // vec3 col = vec3(0.0);  // for(int i = 0; i < 9; i++)  // col += sampleTex[i] \* kernel[i];  // FragColor = vec4(col, 1.0);  } | |

Kết quả



* + 1. Ví dụ 2:

1. Cubemap
2. Geometry shader
   1. Lý thuyết
      * + Nó nằm giữa Vertex shader và fragment shader.
        + Quy tắc truyền dữ liệu in, out giữa các shader là giống hệt nhau. Vert(in,out) -> Geo(in,out) -> frag(in,out)
        + các phương thức, biến ,… khai báo, sử dụng,… tất cả thao tác giống hệt như với vertex shader, fragment shader: ví dụ: normalize(),vec3, vec4,…
        + Khi ở vert gán 1 toạ độ điểm gl\_position -> sau đó toạ độ này sẽ được chuyển vào geometry shader(biến đổi thành 1, nhiều điểm khác hoàn toàn với điểm đưa vào) -> Sau đó nó sẽ đóng gói thành 1 nguyên thuỷ duy nhất -> chuyển qua fragment shader.
        + Khi biến toạ độ nguyên thuỷ từ vertex shader thành một đối tượng nguyên thuỷ khác thì trong geometry ta cần khai báo dòng code sau.

|  |  |  |
| --- | --- | --- |
| **STT** | **Geometry Shader** | **Nội dung** |
| 1 | layout (points, max\_vertices = 1) out; | Từ n điểm gl\_position(vert) chuyển thành 1 điểm khác |
| 2 | layout (line\_strip, max\_vertices = 2) out; | Từ n điểm gl\_Poisition(vert) chuyển thành 1 đường line. |
| 3 | layout (triangle\_strip, max\_vertices = 5) out; | Từ n đểm gl\_position(vert) chuyển thành các triangle(3 điểm thành 1 tam giác) ở đây khai báo max điểm biến thành là 5 điểm |
| 4 |  |  |

Chú ý n điểm này nó sẽ dựa vào kiểu vẽ mà người dùng truyền vào

Trong geoshader dựa vào n này ta có thể gọi gs\_in[0],gs\_in[1],gs\_in[2],…. Số lượng này sẽ dựa vào bảng sau.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Khai báo trong geometry** | **Kiểu vẽ**  **[kieu\_ve]** | **Dạng vẽ(ví dụ)** | **Giá trị n truyền vào geometry shader** |
| 1 | layout (points) in; | GL\_POINTS | glDrawArrays(**[kieu\_ve]**, 0, 4); | 1 |
| 2 | layout (lines) in; | GL\_LINES, GL\_LINE\_STRIP | 2 |
| 3 | layout (lines\_adjacency) in; | GL\_LINES\_ADJACENCY  GL\_LINE\_STRIP\_ADJACENCY | 4 |
| 4 | layout (triangles) in; | GL\_TRIANGLES , GL\_TRIANGLE\_STRIP,  GL\_TRIANGLE\_FAN | 3 |
| 5 | layout (triangles\_adjacency) in; | GL\_TRIANGLES\_ADJACENCY,  GL\_TRIANGLE\_STRIP\_ADJACENCY | 6 |

* + - * Mỗi lần gọi EmitVertex(), vectơ hiện được đặt thành gl\_Position được thêm vào nguyên thủy đầu ra.(từ 1 điểm gl\_position ở vertex shader sau khi gọi lệnh này trong geometry shader thì nó tự động thêm 1 gl\_position ở sau điểm đưa vào)
      * Gọi phương thức EndPrimitive Sẽ gom các EmitVertex() thành 1 nguyên thuỷ đầu ra cho fragment shader.
  1. Các tạo và sử dụng geometry shader
     1. Nguyên lý
        + Khi tạo shaderprogram thay vì ta đính kèm vertext shader và fragment shader thì ta đính kèm thêm file geometry shader.
     2. Các bước sử dụng
        + Bước 1: tạo geometryshader

|  |
| --- |
| #version 330 core  layout (**points**) in; // glDrawArrays(GL\_POINTS, 0, 4); => số lượng n=1=> gl\_in[0]  layout (line\_strip, max\_vertices = 5) out;  // ví dụ chuyển đổi primitive(dựa vào glDrawArrays(..) vẽ) thành đường line  void build\_house(vec4 position){  gl\_Position = position + vec4(-0.2, -0.2,0.0,0.0) ; // 1: bottom-left  EmitVertex();  gl\_Position = position + vec4( 0.2, -0.2,0.0,0.0) ; // 2: bottom-right  EmitVertex();  EndPrimitive();  }  void main(){  build\_house(gl\_in[0].gl\_Position);  } |

* + - * Bước 2: Tạo shaderprogram như bình thường

|  |
| --- |
| shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Resources/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Geometry,":/Resources/obj.geom");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Resources/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log(); |

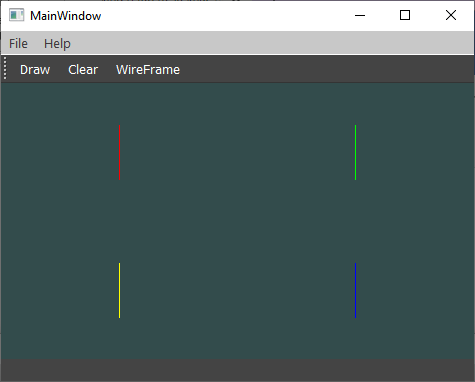
* + - * Bước 3: sử dụng như bình thường không khác gì (khi chỉ sử dụng vertex và fragment)

|  |
| --- |
| shaderProgram.bind();  glBindVertexArray(VAO);  glDrawArrays(GL\_LINE\_STRIP, 0, 4); |

* 1. Ví dụ 1: Cơ bản + Chuyển dữ liệu (vertex – geo – frag)

|  |
| --- |
| float points[] = {  -0.5f, 0.5f,1.0f,0.0f,0.0f, // top-left  0.5f, 0.5f,0.0f,1.0f,0.0f, // top-right  0.5f, -0.5f,0.0f,0.0f,1.0f, // bottom-right  -0.5f, -0.5f,1.0f,1.0f,0.0f // bottom-left  };  void AXBOpenGLWidget::***initializeGL***()  {  *…*  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/Shaders/Resources/Shaders/obj.vert");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Geometry, ":/Shaders/Resources/Shaders/obj.geom");  shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/Shaders/Resources/Shaders/obj.frag");  shaderProgram.*link*();  success = shaderProgram.*link*();  if(!success)  qDebug() << "Error shader: " << shaderProgram.log();  …  glGenBuffers(1, &VBO);  glGenVertexArrays(1, &VAO);  glBindVertexArray(VAO);  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(points), &points, GL\_STATIC\_DRAW);  glEnableVertexAttribArray(0);  glVertexAttribPointer(0, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), 0);  glEnableVertexAttribArray(1);  glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 5 \* sizeof(float), (void\*)(2 \* sizeof(float)));  glBindVertexArray(0);  } |
| void AXBOpenGLWidget::***paintGL***()  {  …  shaderProgram.bind();  glBindVertexArray(VAO);  glDrawArrays(GL\_POINTS, 0, 4);  } |
| // vert  #version 330 core  layout (location = 0) in vec2 aPos;  layout (location = 1) in vec3 aColor;  out VS\_OUT{vec3 color;} vs\_out;  void main()  {  gl\_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);  gl\_PointSize=20;  vs\_out.color= aColor;  } |
| // geo  #version 330 core  layout (points) in;  layout (line\_strip, max\_vertices = 2) out;  in VS\_OUT{  vec3 color;  } gs\_in[];  out vec3 fcolor;  void build\_house(vec4 position1){  fcolor= gs\_in[0].color;  gl\_Position = position1 + vec4(0.0, -0.2,0.0,0.0) ; // 1: bottom-left  EmitVertex();  gl\_Position = position1 + vec4(0.0, 0.2,0.0,0.0) ; // 2: bottom-right  EmitVertex();  EndPrimitive();  }  void main(){  build\_house(gl\_in[0].gl\_Position);  } |
| // frag  #version 330 core  out vec4 FragColor;  in vec3 fcolor;  void main()  {  FragColor = vec4(fcolor, 1.0);  } |

Kết quả

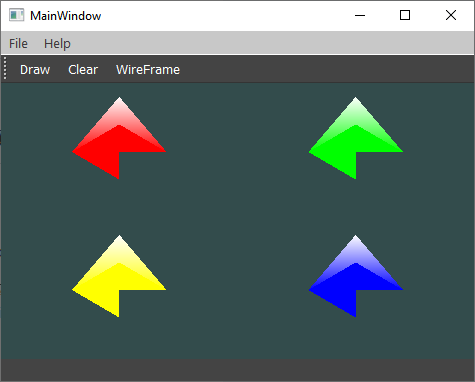


Nếu không có shader thì nó chỉ hiển thị ra 4 điểm bình thường

* 1. Ví dụ 2: Vẽ triangle trong geometry shader

|  |
| --- |
| shaderProgram.bind();  glBindVertexArray(VAO);  glDrawArrays(GL\_POINTS, 0, 4); |
| // vert  #version 330 core  layout (location = 0) in vec2 aPos;  layout (location = 1) in vec3 aColor;  out VS\_OUT{vec3 color;} vs\_out;  void main()  {  gl\_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);  gl\_PointSize=20;  vs\_out.color= aColor;  } |
| // geo  #version 330 core  layout (points) in;  layout (triangle\_strip, max\_vertices = 5) out;  in VS\_OUT{  vec3 color;  } gs\_in[];  out vec3 fcolor;  void build\_house(vec4 position){  fcolor= gs\_in[0].color;  gl\_Position = position + vec4(0.0, -0.2,0.0,0.0) ; // 1: bottom-left  EmitVertex();  gl\_Position = position + vec4(0.0, 0.2,0.0,0.0) ; // 2: bottom-right  EmitVertex();  gl\_Position = position + vec4(-0.2, 0.0,0.0,0.0) ; // 3: Top - left  EmitVertex();  gl\_Position = position + vec4( 0.2, 0.0,0.0,0.0) ; // 4: Top - Right  EmitVertex();  fcolor= vec3(1.0f,1.0f,1.0f);  gl\_Position = position + vec4( 0.0, 0.4,0.0,0.0) ; // 5: Top  EmitVertex();  EndPrimitive();  }  void main(){  build\_house(gl\_in[0].gl\_Position);  } |
| // frag  #version 330 core  out vec4 FragColor;  in vec3 fcolor;  void main()  {  FragColor = vec4(fcolor, 1.0);  } |

Kết quả





1. Instancing
2. Antiasing