**TRƯỜNG ĐẠI HỌC HÀNG HẢI VIỆT NAM  
KHOA CÔNG NGHỆ THÔNG TIN**

**-----\*\*\*-----**

****

**BÁO CÁO BÀI TẬP LỚN**

**HỌC PHẦN “ĐỒ HỌA MÁY TÍNH”**

***Đề tài:***

***XÂY DỰNG TRÒ CHƠI***

***MÁY BẮN BÓNG TENNIS***

***GVHD: ThS. Nguyễn Hạnh Phúc***

***Sinh viên thực hiện: Phạm Quang Cường – Mã sv: 69509***

***Nguyễn Thị Thảo Hiền – Mã sv: 70644***

***Hải Phòng, tháng 04 năm 2019***

**TRƯỜNG ĐẠI HỌC HÀNG HẢI**

**KHOA CÔNG NGHỆ THÔNG TIN**

**BỘ MÔN KHOA HỌC MÁY TÍNH**

**-----\*\*\*-----**

**BÀI TẬP LỚN**

**HỌC PHẦN: ĐỒ HỌA MÁY TÍNH**

**Mã đề tài: 16**

1. **Tên đề tài**

***Xây dựng trò chơi máy bắn bóng tennis.***

1. **Mục đích**

Xây dựng trò chơi máy bắn bóng tennis: Máy bắn bóng mỗi lần bắn ra một quả bóng tennis, người chơi dùng vợt đánh trả lại

1. **Công việc cần thực hiện**

* Tìm hiểu về khái niệm đồ họa máy tính và các khái niệm liên quan đến đồ họa máy tính
* Tìm hiểu về OpenGL và các lập trình trò chơi đơn giản với OpenGL.
* Xây dựng chương trình trò chơi dựa vào các kiến thức đã được học.
* Làm báo cáo bài tập lớn
* Bảo vệ bài tập lớn

1. **Yêu cầu**

* Kết quả: Báo cáo và bảo vệbài tập lớn
  + Hạn nộp báo cáo bài tập lớn:

***Hải Phòng, tháng 04 năm 2017***

NGƯỜI HƯỚNG DẪN

# 

# MỤC LỤC

[MỤC LỤC i](#_Toc7133214)

[GIỚI THIỆU 1](#_Toc7133215)

[CHƯƠNG 1. CƠ SỞ LÝ THUYẾT 2](#_Toc7133216)

[1.1. Các khái niệm cơ bản về đồ họa máy tính 2](#_Toc7133217)

[1.1.1. Định nghĩa Đồ họa máy tính 2](#_Toc7133218)

[1.1.2. Khái niệm OpenGL (Open Graphics Library) 2](#_Toc7133219)

[1.1.3. Các đối tượng đồ họa cơ bản 3](#_Toc7133220)

[1.2. Các phép biến đổi đồ họa cơ sở 4](#_Toc7133221)

[1.2.1. Hệ tọa độ thuần nhất 4](#_Toc7133222)

[1.2.2. Phép tịnh tiến 5](#_Toc7133223)

[1.2.3. Phép quay 5](#_Toc7133224)

[1.2.4. Phép biến đổi tỷ lệ 5](#_Toc7133225)

[1.2.5. Kết hợp các phép biến đổi 6](#_Toc7133226)

[1.3. Vài nét về OpenGL 3.0+ 7](#_Toc7133227)

[CHƯƠNG 2. XÂY DỰNG CHƯƠNG TRÌNH 9](#_Toc7133228)

[2.1. Lớp Shader 9](#_Toc7133229)

[2.2. Lớp Camera 9](#_Toc7133230)

[2.3. Lớp Mesh 10](#_Toc7133231)

[2.4. Lớp Model 10](#_Toc7133232)

[CHƯƠNG 3. CÀI ĐẶT CÁC LỚP 11](#_Toc7133233)

[3.1. Lớp Shader 11](#_Toc7133234)

[3.2. Lớp Camera 12](#_Toc7133235)

[3.3. Lớp Mesh 14](#_Toc7133236)

[3.4. Lớp Model 15](#_Toc7133237)

[3.5. Hàm chính (main()) 18](#_Toc7133239)

[CHƯƠNG 3. KẾT QUẢ CHƯƠNG TRÌNH 23](#_Toc7133240)

[KẾT LUẬN VÀ KIẾN NGHỊ 25](#_Toc7133241)

[TÀI LIỆU THAM KHẢO 26](#_Toc7133242)

# GIỚI THIỆU

Trước kia, khi đồ họa chưa phát triển, các ứng dụng và thậm chí là trò chơi được xây dựng trên các màn hình console với các dòng có kích thước cố định. Điều này làm trở ngại trong quá trình làm việc và giải trí vì rất khó để hiển thị một hình ảnh có kích thước và nội dung bất kỳ. Đồ họa máy tính là phương pháp giải quyết vấn đề này. Khi dùng đồ họa, hình ảnh được hiển thị trực quan và sống động.

Đồ họa máy tính ảnh hưởng đến mọi khía cạnh cuộc sống. Ví dụ, các ứng dụng tương tác người dùng tốt hơn, các phần mềm mô phỏng để tìm hiểu trước khi bắt đầu làm các mô hình thực, phim ảnh, …

OpenGL là môi trường phát triển các ứng dụng đồ họa 2D và 3D được lần đầu giới thiệu vào năm 1992. OpenGL ra đời thúc đẩy quá trình phát triển của hàng loạt ứng dụng máy tính sử dụng đồ họa và nâng cao chất lượng hiển thị

# CHƯƠNG 1. CƠ SỞ LÝ THUYẾT

## Các khái niệm cơ bản về đồ họa máy tính

### Định nghĩa Đồ họa máy tính

Đồ họa máy tính là tập hợp các công việc dùng máy tính để tạo ra hình ảnh: tạo, xử lý, và lưu trữ các mô hình.

Một hệ thống đồ họa thường gồm 2 thành phần:

* Phần cứng: Các thiết bị hiển thị hình ảnh: Màn hình, thiết bị hiển thị pixel, máy in, các hệ màu, không gian màu, các thiết bị nhập
* Phần mềm:
* Các công cụ lập trình các ứng dụng đồ họa
* Các ứng dụng được lập trình sẵn cho việc thiết kế đồ họa

### Khái niệm OpenGL (Open Graphics Library)

OpenGL là một tiêu chuẩn kĩ thuật đồ họa nhằm tạo ra các giao diện ứng dụng không gian 3 chiều. OpenGL được sử dụng rộng rãi trên hầu hết các máy tính, đặc biệt là đối với các game.

Ứng dụng của OpenGL rất lớn, với khoảng hơn 250 hàm trong việc lập trình thay vì dùng các đối tượng cơ bản.

### Các đối tượng đồ họa cơ bản

Bất kỳ một đối tượng đồ họa nào cũng được tạo thành từ các đối tượng cơ sở (primitive). Ví dụ như một mô hình máy bay được cấu tạo từ các bộ phận như cánh, buồng lái, … và các bộ phận lại được cấu tạo từ các chi tiết nhỏ hơn như các điểm, đoạn thẳng, vòng cung, xoắn ốc, …

Các đối tượng cơ bản gồm:

* Điểm
* Đoạn thẳng
* Đường gấp khúc
* Vùng tô
* Chuỗi, kí tự
* …

Để vẽ các đối tượng đồ họa cơ bản sử dụng thư viện OpenGL, ta sử dụng các câu lệnh

glBegin(chế\_độ\_vẽ);

glVertex2i(x,y);

glEnd();

Trong đó, các chế độ vẽ gồm GL\_TRIANGLES, GL\_POINTS, GL\_LINES, GL\_QUAD, GL\_POLYGON, … glVertex2i(x,y) là danh sách các tọa độ của đối tượng đồ họa đó. glEnd() để kết thúc chế độ vẽ.

## Các phép biến đổi đồ họa cơ sở

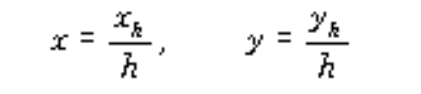
Các phép biến đổi đồ họa cho phép chúng ta dễ dàng thao tác trên các đối tượng đồ họa như thay đổi tọa độ của các đối tượng, kích thước, hướng và hình dạng cảu chúng

Các phép biến đổi đồ họa cơ sở gồm:

* Phép tịnh tiến
* Phép quay
* Phép tỉ lệ
* Ngoài ra còn có các phép đối xứng và biến dạng

### Hệ tọa độ thuần nhất

Để dễ dàng trong quá trình biểu diễn và chuyển đổi, người ta thường đưa tọa độ các điểm về hệ tọa độ thuần nhất. Ví dụ một điểm P(x, y) có tọa độ là (x, y) trên tọa dộ phẳng có tọa độ (xh, yh, h) trong hệ tọa độ thuần nhất. Trong đó, xh, yh và h là các tọa độ tỉ lệ. Ta có công thức:

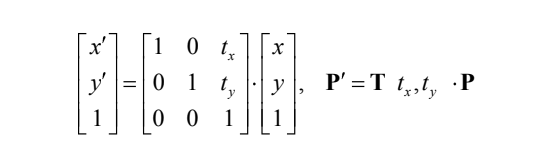
****

Để đơn giản ta thường chọn h = 1, lúc này điểm P(x,y) trong hệ tọa độ thuần nhất là P(x, y, 1).

Một điểm được biến đổi bằng các thay đổi tọa độ của chúng theo phép biến đổi. Ví dụ: đối với phép tịnh tiến, để biến đổi chúng ta sẽ cộng vào điểm độ dời vào tọa độ. Ngoài ra chúng ta còn có một phương pháp đơn giản, hiệu quả, và nhanh chóng hơn, đó là nhân điểm đó với ma trận của phép biến đổi.

### Phép tịnh tiến

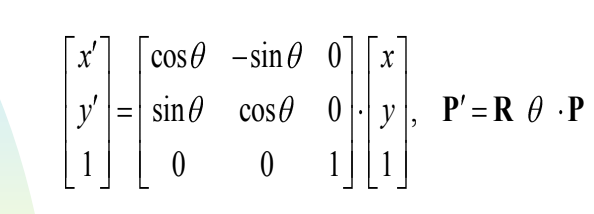
Để tịnh tiến một điểm có tọa độ P(x, y) sang một tọa độ mới với độ dời là *tx* và *ty*, chúng ta nhân điểm đó với ma trận của phép tịnh tiến:



Kết quả thu được điểm P’(x’,y’).

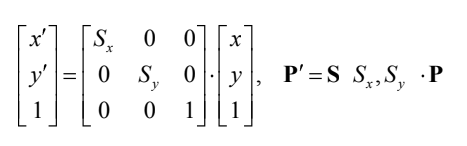
### Phép quay

Tương tự đối với phép tịnh tiến, để quay một điểm P(x, y) ngược chiều kim đồng hồ ta có ma trận:



\

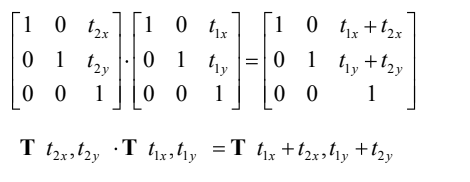
### Phép biến đổi tỷ lệ



### Kết hợp các phép biến đổi

Khi muốn thực hiện các phép biến đổi liên tiếp, ví dụ tịnh tiến, phóng to, và quay một góc, ta sử dụng kết hợp các phép biến đổi bằng cách nhân lần lượt các phép biến đổi với nhau.

Ví dụ để thực hiện 2 phép tịnh tiến một lúc ta nhân 2 ma trận tịnh tiến với nhau:

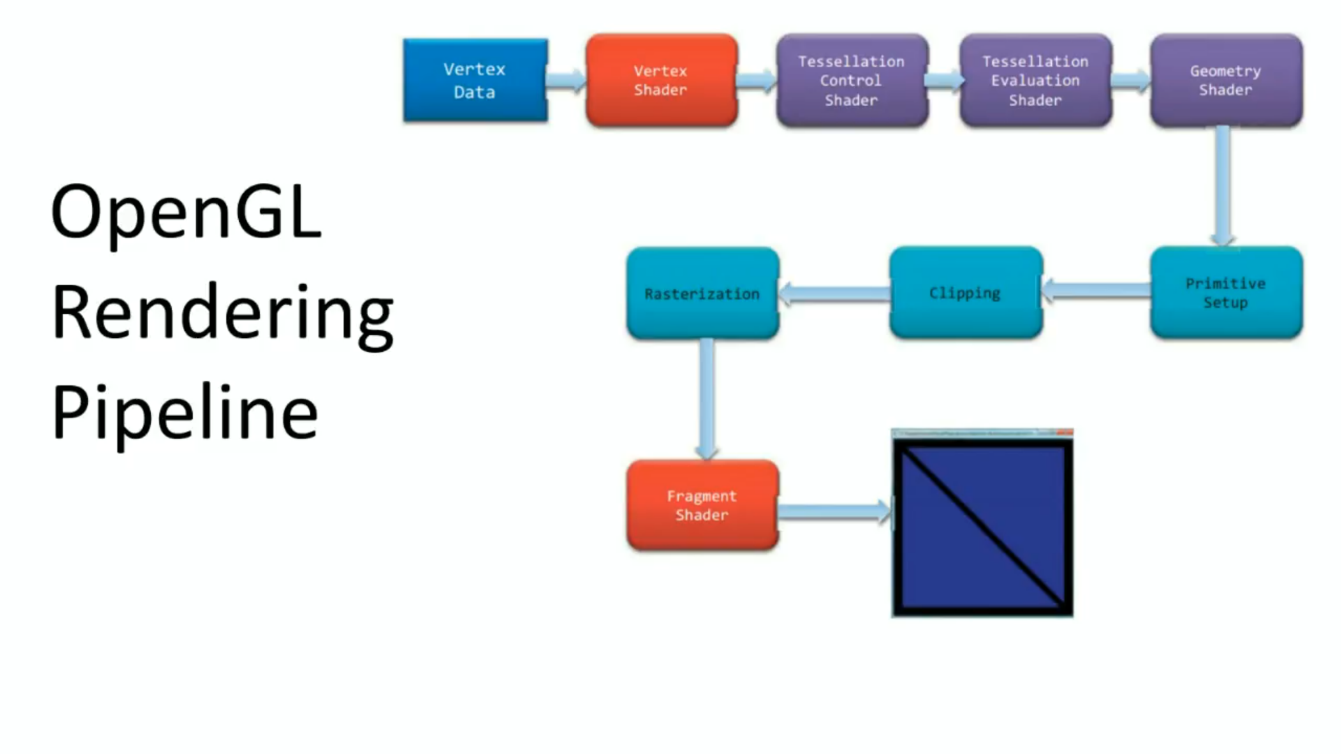


Chú ý: Phép nhân ma trận không có tính chất giao hoán. Ma trận của phép biến đổi bên phải sẽ được thực hiện trước.

## Vài nét về OpenGL 3.0+

OpenGL có nhiều phiên bản, trong đó gần đây nhất là phiên bản 4.6. Các phiên bản gần đây được viết lại một nhiều hàm và các thức kiết xuất.

Đối với OpenGL 3.0 trở lên, kiết xuất hình ảnh trải qua 9 giai đoạn:



Trong đó có 4 bước mà chúng ta có thể lập trình được thông qua các “Shader”. Shader là một đoạn mã được viết bằng GLSL (một ngôn ngữ lập trình gần giống C)

* Vertex shader: Xử lý thông tin tọa độ các đỉnh
* Tesselation: Chia dữ liệu vào các đối tượng độ họa cơ sở
* Geometry shader: Xử lý các nhóm tọa độ và đưa ra các tọa độ mới
* Fragment shader: Xử lý dữ liệu của từng Pixel như màu, …

Trong chương trình OpenGL thường có Fragment Shader và Vertex Shader, ví dụ:

***vertex shader example***

#version 330

layout (location = 0) in vec3 pos;

void main

{

gl\_Position = vec4(pos , 1.0);

}

***fragment shader example***

#version 330

out vec4 colour;

void main()

{

colour=vec4(1.0 , 0.0, 0.0 , 1.0);

}

# CHƯƠNG 2. XÂY DỰNG CHƯƠNG TRÌNH

Để xây dựng chương trình ta cần xây dựng các thành phần sau:

* Lớp Shader: dùng để biên dịch và tạo ra các chương trình.
* Lớp Camera: dùng để tạo và thay đổi khung nhìn.
* Lớp Model: dùng để tải các mẫu vật.
* Lớp Mesh: dùng để lưu tọa độ các đỉnh, ảnh texure, chỉ số vẽ

## Lớp Shader

* Tên lớp: Shader
* Thuộc tính:
  + GLuint Program;
* Phương thức:
  + Khởi tạo :Shader();
  + Sử dụng :void Use();

## Lớp Camera

* Tên lớp: Camera
* Thuộc tính:
  + Postion : vec3
  + Front : vec3
  + Up : vec3
  + Right : vec3
  + worldUp : vec3
  + yaw : GLfloat
  + pitch : Glfloat
  + movementSpeed : GLfloat
* Phương thức:
  + Khởi tạo : Camera()
  + Lấy ma trận view : mat4 getViewMatrix
  + Xử lý bàn phím : void ProcesssKeyBoard
  + Lấy vị trí Camera : vec3 getCameraPosition
  + Cập nhật : void updateCamera()

## Lớp Mesh

* Tên lớp: Mesh
* Thuộc tính:
  + VAO : GLint
  + VBO : GLint
  + EBO : GLint
* Phương thức:
  + Khởi tạo : Mesh()
  + SetUpMesh : void SetUpMesh()
  + Draw : void Draw()

## Lớp Model

* Tên lớp: Model
* Thuộc tính:
  + meshes : vector<mesh>
  + directory : string
  + textures\_loaded :vector<Texture>
* Phương thức:
  + Khỏi tạo : Model()
  + Draw :void Draw();
  + loadModel : void loadModel();
  + TextureFromFile :GLint TextureFromFile()
  + loadMaterialTextures :vector<Texture>loadMaterialTextures()
  + processMesh :Mesh processMesh()

# CHƯƠNG 3. CÀI ĐẶT CÁC LỚP

## Lớp Shader

class Shader

{

public:

GLuint Program;

void Use()

{

glUseProgram(this->Program);

}

Shader(const GLchar \*vertexPath, const GLchar \*fragmentPath)

{

std::string vertexCode;

std::string fragmentCode;

std::ifstream vShaderFile;

std::ifstream fShaderFile;

vShaderFile.open(vertexPath);

fShaderFile.open(fragmentPath);

std::stringstream vShaderStream, fShaderStream;

vShaderStream << vShaderFile.rdbuf();

fShaderStream << fShaderFile.rdbuf();

vertexCode = vShaderStream.str();

fragmentCode = fShaderStream.str();

const GLchar \*vShaderCode = vertexCode.c\_str();

const GLchar \*fShaderCode = fragmentCode.c\_str();

GLuint vertex, fragment;

GLint success;

GLchar infoLog[512];

vertex = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertex, 1, &vShaderCode, NULL);

glCompileShader(vertex);

glGetShaderiv(vertex, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(vertex, 512, NULL, infoLog);

std::cout << "VERTEX\_SHADER\_COMPILE\_FAIL\n" << infoLog <<

std::endl;

}

fragment = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragment, 1, &fShaderCode, NULL);

glCompileShader(fragment);

glGetShaderiv(fragment, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(fragment, 512, NULL, infoLog);

std::cout << "FRAGMENT\_SHADER\_COMPILE\_FAIL\n" << infoLog <<

std::endl;

}

this->Program = glCreateProgram();

glAttachShader(this->Program, vertex);

glAttachShader(this->Program, fragment);

glLinkProgram(this->Program);

}

};

## Lớp Camera

enum CameraMove {

forward,

backward,

left,

right

};

class Camera {

private:

vec3 position;

vec3 front;

vec3 up;

vec3 right;

vec3 worldUp;

GLfloat Yaw;

GLfloat pitch;

GLfloat movementSpeed;

public:

Camera( vec3 position = vec3(0, 0, 0), vec3 up = vec3(0, 1, 0),

GLfloat Yaw = YAW, GLfloat Pitch = PITCH) :front( vec3(0, 0, -1)),

movementSpeed(SPEED)

{

this->position = position;

this->worldUp = up;

this->Yaw = Yaw;

this->pitch = Pitch;

this->updateCamera();

}

mat4 getViewMatrix()

{

return lookAt(position, position + front, up);

}

void HandleKeyBoard(CameraMove direction, GLfloat deltaTime)

{

GLfloat velocity = this->movementSpeed\*deltaTime;

if (CameraMove::left == direction)

{

this->position -= this->right\*velocity;

}

if (CameraMove::right == direction)

{

this->position += this->right\*velocity;

}

if (CameraMove::forward == direction)

{

this->position += this->front\*velocity;

}

if (CameraMove::backward == direction)

{

this->position -= this->front\*velocity;

}

}

void updateCamera()

{

vec3 front;

front.x = cos( radians(this->Yaw))\*cos( radians(this->pitch));

front.y = sin( radians(this->pitch));

front.z = sin( radians(this->Yaw))\*cos( radians(this->pitch));

this->front = normalize(front);

this->right = normalize( cross(this->front, this->worldUp));

this->up = normalize( cross(this->right, this->front));

}

};

## Lớp Mesh

class Mesh

{

private:

GLuint VAO, VBO, EBO;

public:

vector<Vertex> vertices;

vector<GLuint> indices;

vector<Texture> textures;

Mesh(vector<Vertex> vertices,

vector<GLuint> indices,

vector<Texture> textures)

{

this->vertices = vertices;

this->indices = indices;

this->textures = textures;

this->SetUpMesh();

}

void Draw(Shader shader)

{

GLuint diffuseNr = 1;

GLuint specularNr = 1;

for (int i = 0; i < this->textures.size(); i++)

{

glActiveTexture(GL\_TEXTURE0 + i);

stringstream ss;

string number;

string name = this->textures[i].type;

if ("texture\_diffuse" == name)

{

ss << diffuseNr++;

}

else if ("texture\_specular" == name)

{

ss << specularNr++;

}

number = ss.str();

glUniform1i(glGetUniformLocation(shader.Program, (name +

number).c\_str()),i);

glBindTexture(GL\_TEXTURE\_2D, this->textures[i].id);

}

glUniform1f(glGetUniformLocation(shader.Program,

"material.shininess"), 32);

glBindVertexArray(this->VAO);

glDrawElements(GL\_TRIANGLES, this->indices.size(), GL\_UNSIGNED\_INT, 0);

glBindVertexArray(0);

for (int i = 0; i < this->textures.size(); i++)

{

glActiveTexture(GL\_TEXTURE0 + i);

glBindTexture(GL\_TEXTURE\_2D, 0);

}

}

void SetUpMesh()

{

glGenVertexArrays(1, &this->VAO);

glGenBuffers(1, &VBO);

glGenBuffers(1, &EBO);

glBindVertexArray(this->VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, this->VBO);

glBufferData(GL\_ARRAY\_BUFFER, this->vertices.size() \* sizeof(Vertex), &this->vertices[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, this->EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, this->indices.size()\*sizeof(GLuint),&this->indices[0],GL\_STATIC\_DRAW);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (GLvoid \*)0);

glEnableVertexAttribArray(1);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (GLvoid \*)offsetof(Vertex, normal));

glEnableVertexAttribArray(2);

glVertexAttribPointer(2, 3, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (GLvoid\*)offsetof(Vertex, texCoord));

glBindVertexArray(0);

}

};

## Lớp Model

class Model

{

private:

string directory;

vector<Texture> textures\_loaded;

vector<Mesh> meshes;

public:

Model( const GLchar \*path )

{

this->loadModel( path );

}

void Draw\_Model( Shader shader )

{

for ( GLuint i = 0; i < this->meshes.size( ); i++ )

{

this->meshes[i].Draw( shader );

}

}

vector<Texture> loadMaterialTextures(aiMaterial \*mat, aiTextureType type,

string typeName)

{

vector<Texture> textures;

for (GLuint i = 0; i < mat->GetTextureCount(type); i++)

{

aiString str;

mat->GetTexture(type, i, &str);

GLboolean skip = false;

for (GLuint j = 0; j < textures\_loaded.size(); j++)

{

if (textures\_loaded[j].path == str)

{

textures.push\_back(textures\_loaded[j]);

skip = true;

break;

}

}

if (!skip)

{

Texture texture;

texture.id = LoadTexture(str.C\_Str(), this->directory);

texture.type = typeName;

texture.path = str;

textures.push\_back(texture);

this->textures\_loaded.push\_back(texture);

}

}

return textures;

}

void processNode( const aiScene\* scene, aiNode\* node )

{

for ( GLuint i = 0; i < node->mNumMeshes; i++ )

{

aiMesh\* mesh = scene->mMeshes[node->mMeshes[i]];

this->meshes.push\_back( this->processMesh( scene,mesh ) );

}

for ( GLuint i = 0; i < node->mNumChildren; i++ )

{

this->processNode( scene,node->mChildren[i] );

}

}

Mesh processMesh( const aiScene \*scene , aiMesh \*mesh)

{

vector<GLuint> indices;

vector<Texture> textures;

vector<Vertex> vertices;

for ( GLuint i = 0; i < mesh->mNumVertices; i++ )

{

Vertex vertex;

glm::vec3 vector;

vector.x = mesh->mVertices[i].x;

vector.y = mesh->mVertices[i].y;

vector.z = mesh->mVertices[i].z;

vertex.position = vector;

vector.x = mesh->mNormals[i].x;

vector.y = mesh->mNormals[i].y;

vector.z = mesh->mNormals[i].z;

vertex.normal = vector;

if( mesh->mTextureCoords[0] )

{

glm::vec2 vec;

vec.x = mesh->mTextureCoords[0][i].x;

vec.y = mesh->mTextureCoords[0][i].y;

vertex.texCoord = vec;

}

else

{

vertex.texCoord = glm::vec2(0.0f, 0.0f);

}

vertices.push\_back( vertex );

}

for ( GLuint i = 0; i < mesh->mNumFaces; i++ )

{

aiFace face = mesh->mFaces[i];

for ( GLuint j = 0; j < face.mNumIndices; j++ )

{

indices.push\_back( face.mIndices[j] );

}

}

if( mesh->mMaterialIndex >= 0 )

{

aiMaterial\* material = scene->mMaterials[mesh->mMaterialIndex];

vector<Texture> diffuseMaps = this->loadMaterialTextures( material,

aiTextureType\_DIFFUSE, "texture\_diffuse" );

textures.insert( textures.end( ), diffuseMaps.begin( ), diffuseMaps.end( ) );

vector<Texture> specularMaps = this->loadMaterialTextures( material,

aiTextureType\_SPECULAR, "texture\_specular" );

textures.insert( textures.end( ), specularMaps.begin( ),

specularMaps.end( ) );

}

return Mesh( vertices, indices, textures );

}

void loadModel(string path)

{

Assimp::Importer importer;

const aiScene \*scene = importer.ReadFile(path, aiProcess\_Triangulate |

aiProcess\_FlipUVs);

if (!scene || scene->mFlags == AI\_SCENE\_FLAGS\_INCOMPLETE ||

!scene->mRootNode)

{

cout << "Loi ASSIMP: " << importer.GetErrorString() << endl;

return;

}

this->directory = path.substr(0, path.find\_last\_of('/'));

this->processNode(scene, scene->mRootNode);

}

};

GLint LoadTexture( const char \*path, string directory )

{

string filename = string( path );

filename = directory + '/' + filename;

GLuint textureID;

glGenTextures( 1, &textureID );

int width, height;

unsigned char \*image = SOIL\_load\_image( filename.c\_str( ), &width, &height, 0, SOIL\_LOAD\_RGB );

glBindTexture( GL\_TEXTURE\_2D, textureID );

glTexImage2D( GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, image );

glGenerateMipmap( GL\_TEXTURE\_2D );

glTexParameteri( GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT );

glTexParameteri( GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT );

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

glTexParameteri( GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glBindTexture( GL\_TEXTURE\_2D, 0 );

SOIL\_free\_image\_data( image );

return textureID;

## Hàm chính (main())

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include"glut.h"

#include "Shader.h"

#include"SOIL2/SOIL2.h"

#include<glm/glm.hpp>

#include<glm/gtc/matrix\_transform.hpp>

#include<glm/gtc/type\_ptr.hpp>

#include"Camera.h"

#include"Mesh.h"

#include"Model.h"

using namespace std;

int SCREEN\_WIDTH = 800, SCREEN\_HEIGHT = 600;

bool keys[1024];

bool FirstMouse = true;

void drawBall(Model ball, Shader shader);

void keyCallBack(GLFWwindow \* window, int key, int scancode, int action, int mode);

void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods);

void DoMovement();

Camera camera(glm::vec3(0, -0.5, 23));

GLfloat lastX = SCREEN\_WIDTH / 2;

GLfloat lastY = SCREEN\_WIDTH / 2;

GLfloat deltaTime = 0;

GLfloat lastTime = 0;

glm::mat4 view(1), projection(1), model(1);

glm::mat4 x(1);

float moveX = 0, moveY = 0, spin = 0, moveJkt = 0, hitSpin = 0, ballMoveX = 0, ballMoveZ = 0, ballMoveY = 0, h = 20;

int i = 0;

int point = 0, live = 5;

bool isUp = true;

float xOffset = 0, zOffset = 4;

int main()

{

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

glfwWindowHint(GLFW\_RESIZABLE, GL\_FALSE);

GLFWwindow\* window = glfwCreateWindow(SCREEN\_WIDTH, SCREEN\_WIDTH,

"BTL", nullptr, nullptr);

glfwMakeContextCurrent(window);

glfwGetFramebufferSize(window, &SCREEN\_WIDTH, &SCREEN\_HEIGHT);

glfwSetKeyCallback(window, keyCallBack);

glfwSetMouseButtonCallback(window, mouse\_button\_callback);

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

glewExperimental = GL\_TRUE;

glewInit();

glViewport(0, 0, SCREEN\_WIDTH, SCREEN\_HEIGHT);

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

Shader shader("modelLoading.vs", "modelLoading.frag");

Shader textShader("textShader.vert", "textShader.frag");

Model ourModel("res/10093\_Wembley\_stadion\_V3\_Iteration0.obj");

Model charecter("res/2nrtbod1out.obj");

Model tennis\_jacket("res/10540\_Tennis\_racket\_V2\_L3.obj");

Model ball("res/10539\_tennis\_ball\_L3.obj");

projection = glm::perspective(45.0f, (GLfloat)SCREEN\_WIDTH /

(GLfloat)SCREEN\_HEIGHT, 0.1f, 100.0f);

while (!glfwWindowShouldClose(window))

{

GLfloat currentFrame = glfwGetTime();

deltaTime = currentFrame - lastTime;

lastTime = currentFrame;

glfwPollEvents();

DoMovement();

glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

shader.Use();

view = camera.getViewMatrix();

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "projection"),

1, GL\_FALSE, glm::value\_ptr(projection));

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "view"), 1,

GL\_FALSE, glm::value\_ptr(view));

glm::mat4 model(1);

model = glm::translate(model, glm::vec3(0.0f, -21, 0));

model = glm::scale(model, glm::vec3(0.006));

model = glm::rotate(model, glm::radians(-90.0f), glm::vec3(1, 0, 0));

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "model"), 1,

GL\_FALSE, glm::value\_ptr(model));

ourModel.Draw\_Model(shader);

glm::mat4 x(1);

x = glm::translate(x, glm::vec3(moveX, -2, moveY + 20));

x = glm::scale(x, glm::vec3(0.006));

x = glm::rotate(x, glm::radians(-90.0f), glm::vec3(1, 0, 0));

x = glm::rotate(x, glm::radians(180.0f + spin + hitSpin), glm::vec3(0,

0, 1));

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "model"), 1,

GL\_FALSE, glm::value\_ptr(x));

charecter.Draw\_Model(shader);

glm::mat4 y(1);

y = glm::translate(y, glm::vec3(moveX + 0.45 + moveJkt, -1.45, moveY +

19.6));

y = glm::scale(y, glm::vec3(0.01));

y = glm::rotate(y, glm::radians(-60.0f), glm::vec3(1, 1, 0));

y = glm::rotate(y, glm::radians(hitSpin), glm::vec3(0, 0, 1));

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "model"), 1,

GL\_FALSE, glm::value\_ptr(y));

tennis\_jacket.Draw\_Model(shader);

drawBall(ball, shader);

if (live == 0) glfwSetWindowShouldClose(window, GL\_TRUE);

spin = 0;

glfwSwapBuffers(window);

}

}

float rand\_FloatRange(float a, float b)

{

return ((b - a) \* ((float)rand() / RAND\_MAX)) + a;

}

void drawBall(Model ball, Shader shader)

{

if (isUp)

{

ballMoveX += xOffset;

ballMoveZ += zOffset;

ballMoveY -= 2 - 0.03\*(20 - h);

}

if (!isUp)

{

ballMoveX += xOffset;

ballMoveZ += zOffset;

ballMoveY += 2 - 0.03\*(20 - h);

}

if (ballMoveY <= -55.0f)

{

isUp = false;

}

h -= 0.2;

if (ballMoveY >= h)

{

ballMoveZ++;

isUp = true;

}

glm::mat4 z(1);

z = glm::scale(z, glm::vec3(0.04));

z = glm::translate(z, glm::vec3(ballMoveX, ballMoveY, ballMoveZ));

glUniformMatrix4fv(glGetUniformLocation(shader.Program, "model"), 1,

GL\_FALSE, glm::value\_ptr(z));

cout << ballMoveX << " " << ballMoveY << " " << ballMoveZ << " " << moveX

<< endl;

//reset ball

if (zOffset < 0 && ballMoveY < -55.6)

{

ballMoveX = ballMoveY = ballMoveZ = 0;

zOffset = -zOffset;

isUp = true;

h = 20;

point++;

xOffset = rand\_FloatRange(-1.2, 1.2);

cout << 1;

}

if (zOffset > 0 && ballMoveY < -55.6)

{

ballMoveX = ballMoveY = ballMoveZ = 0;

zOffset++;

isUp = true;

h = 20;

live--;

xOffset = rand\_FloatRange(-1.2, 1.2);

cout << 1;

}

ball.Draw\_Model(shader);

}

void keyCallBack(GLFWwindow \* window, int key, int scancode, int action, int mode)

{

if (key == GLFW\_KEY\_ESCAPE && action == GLFW\_PRESS)

{

glfwSetWindowShouldClose(window, GL\_TRUE);

}

if (key >= 0 && key < 1024)

{

if (GLFW\_PRESS == action)

{

keys[key] = true;

}

else

if (GLFW\_RELEASE == action)

{

keys[key] = false;

}

}

}

void mouse\_button\_callback(GLFWwindow\* window, int button, int action, int mods)

{

if (button == GLFW\_MOUSE\_BUTTON\_LEFT)

{

if ((action == GLFW\_PRESS || action == GLFW\_REPEAT) &&

(ballMoveX / 20 > moveX - 1 && ballMoveX / 20 < moveX + 1))

{

hitSpin = 45;

moveJkt = -0.3;

zOffset += 1;

zOffset = -zOffset;

}

if (action == GLFW\_RELEASE)

{

hitSpin = 0;

moveJkt = 0;

}

}

}

void DoMovement()

{

GLfloat velocity = 6 \* deltaTime;

if (keys[GLFW\_KEY\_W] || keys[GLFW\_KEY\_UP])

{

camera.HandleKeyBoard(CameraMove::forward, deltaTime);

moveY -= velocity;

hitSpin = 0;

moveJkt = 0;

}

if (keys[GLFW\_KEY\_S] || keys[GLFW\_KEY\_DOWN])

{

camera.HandleKeyBoard(CameraMove::backward, deltaTime);

moveY += velocity;

hitSpin = 0;

moveJkt = 0;

}

if (keys[GLFW\_KEY\_A] || keys[GLFW\_KEY\_LEFT])

{

camera.HandleKeyBoard(CameraMove::left, deltaTime);

moveX -= velocity;

hitSpin = 45;

moveJkt = -0.3;

}

if (keys[GLFW\_KEY\_D] || keys[GLFW\_KEY\_RIGHT])

{

camera.HandleKeyBoard(CameraMove::right, deltaTime);

moveX += velocity;

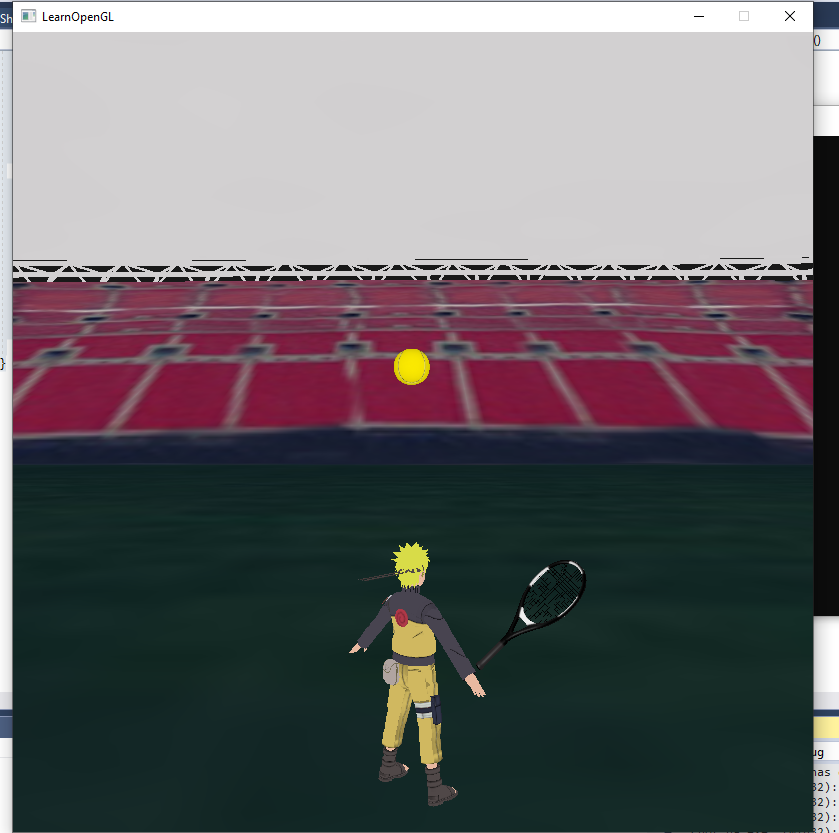
hitSpin = -45;

moveJkt = 0.03;

}

}

# CHƯƠNG 3. KẾT QUẢ CHƯƠNG TRÌNH





## KẾT LUẬN

Chúng em đã cố gắng thực hiện yêu cầu của bài toán đặt ra, tuy nhiên có một số vấn để: chưa hiển thị được điểm số người chơi, số lượt chơi còn lại, chương trình chưa logic, một số lỗi hiển thị.

Chúng em sẽ cố gắng tìm hiểu để hoàn thiện các lỗi trên. Cảm ơn thầy cô và các bạn đã theo dõi.

# TÀI LIỆU THAM KHẢO

1) **Đồ họa máy tính** , Nhà xuất bản Đại học Bưu Chính Viễn Thông.

2) <https://www.youtube.com/watch?v=yQx_pMsYqzU>