**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**



**ĐỒ HỌA MÁY TÍNH - CQ2016/2**

*2018 – 2019*

**­­**

|  |  |
| --- | --- |
|  | **BÁO CÁO ĐỒ ÁN LÝ THUYẾT** |
|  |  |
|  |  |

***Tp.Hồ Chí Minh, 27/12/2018***

**1612406\_1612425\_1612431**

# MỤC LỤC

[MỤC LỤC HÌNH ẢNH 3](#_Toc534230807)

[THÔNG TIN NHÓM 3](#_Toc534230808)

[A. Tên đồ án môn học 4](#_Toc534230809)

[B. Motivation 4](#_Toc534230810)

[C. Problem Statement 5](#_Toc534230811)

[D. Technical Report 5](#_Toc534230812)

[I. Cách sử dụng OpenGL trong các tác vụ 3D (tĩnh và động)[1] [2] [7] 5](#_Toc534230813)

[1. Tổng quan về OpenGL 5](#_Toc534230814)

[1.1. Open GL là gì? [6] 5](#_Toc534230815)

[1.2. Mô hình hoạt động của OpenGL 10](#_Toc534230816)

[1.3. Cấu trúc lệnh trong OpenGL 11](#_Toc534230817)

[1.4. OpenGL Utility Toolkit (GLUT) 11](#_Toc534230818)

[2. Vẽ các đối tượng hình học cơ bản 12](#_Toc534230819)

[2.1. Một số thao tác cơ bản 12](#_Toc534230820)

[2.2. Vẽ các đối tượng hình học 13](#_Toc534230821)

[3. Viewing 15](#_Toc534230822)

[3.1. Giới thiệu 15](#_Toc534230823)

[3.2. Thao tác trên ModelView 17](#_Toc534230824)

[3.3. Thao tác trên Projection (Phép chiếu) 19](#_Toc534230825)

[3.4. Thao tác trên Viewport 21](#_Toc534230826)

[4. Chế độ màu 22](#_Toc534230827)

[4.1. Chế độ màu RGBA 22](#_Toc534230828)

[4.2. Thiết lập mô hình shading 23](#_Toc534230829)

[5. Chiếu sáng 26](#_Toc534230830)

[5.1. Các loại nguồn sáng 26](#_Toc534230831)

[5.2. Tạo nguồn sáng 26](#_Toc534230832)

[5.3. Tính chất vật liệu 28](#_Toc534230833)

[6. Pha trộn, giảm hiệu ứng răng cưa, sương mù 29](#_Toc534230834)

[6.1. Pha trộn 29](#_Toc534230835)

[6.2. Giảm hiệu ứng răng cưa 31](#_Toc534230836)

[6.3. Sương mù 33](#_Toc534230837)

[7. Display Lists 35](#_Toc534230838)

[7.1. Giới thiệu 35](#_Toc534230839)

[7.2. Thao tác với Display List 36](#_Toc534230840)

[8. Dán Texture 37](#_Toc534230841)

[8.1. Texture là gì? 37](#_Toc534230842)

[8.2. Cách phủ texture 38](#_Toc534230843)

[9. Framebuffer 42](#_Toc534230844)

[10. Quadrics 43](#_Toc534230845)

[11. Chọn đối tượng 44](#_Toc534230846)

[13.1. Giới thiệu 44](#_Toc534230847)

[13.2. Các thao tác trên chế độ selection 44](#_Toc534230848)

[II. Xây dựng ứng dụng 47](#_Toc534230849)

[1. Giới thiệu 47](#_Toc534230850)

[2. Phát biểu bài toán 47](#_Toc534230851)

[3. Giải quyết bài toán 48](#_Toc534230852)

[4. Thực nghiệm 50](#_Toc534230853)

[4.1. Giao diện 51](#_Toc534230854)

[4.2. Cài đặt chi tiết 53](#_Toc534230855)

[5. Kết luận 60](#_Toc534230856)

[E. Experimental Results 60](#_Toc534230857)

[F. References 61](#_Toc534230858)

# MỤC LỤC HÌNH ẢNH

[Figure 1. Solar System 51](#_Toc534230741)

[Figure 2. Thông tin Solar System 52](#_Toc534230742)

[Figure 3. Thông tin Trái Đất 53](#_Toc534230743)

# THÔNG TIN NHÓM

|  |  |
| --- | --- |
| **MSSV** | **Họ và tên** |
| 1612406 | Đặng Phương Nam |
| 1612425 | Tạ Đăng Hiếu Nghĩa |
| 1612431 | Trần Bá Ngọc |

# Tên đồ án môn học

|  |
| --- |
| Xây dựng ứng dụng 3D (tĩnh và động) dựa vào OpenGL trên môi trường Windows.  Trình bày cách sử dụng OpenGL trong các tác vụ 3D (tĩnh và động) + xây dựng ứng dụng. |

# Motivation

Hiện nay, OpenGL được áp dụng rộng rãi trong các game hỗ trợ đồ họa 3 chiều và 2 chiều. Nhờ sự đơn giản của nó, người thiết kế có thể vẽ các cảnh trong game tưởng chừng rất khó khăn chỉ với các hàm cơ bản. Bạn là một game thủ chuyên nghiệp, một designer thường xuyên tiếp xúc với các phần mềm đồ họa như AutoCad, CorelDRAW thì việc tìm hiểu ý nghĩa và tính năng mà OpenGL mang lại là vô cùng cần thiết.



|  |
| --- |
| **Các trò chơi được viết với OpenGL** |
| .America's Army .BaldersGate 2 – Mặc định dùng Direct3D .Call of Duty .City of Heroes .City of Villains .CounterStrike 1.6 .Doom 3 .ETQW .Half-Life .Neverwinter Nights .Quake .Serious Sam .Serious Sam SE .Serious Sam 2 – Mặc định dùng Direct3D .Unreal .Warcraft 3 – Mặc định dùng Direct3D .World of Warcraft - OpenGL trên Mac .HomeWorld 2 .Minecraf |

Ngoài ra OpenGL còn dùng trong các ứng dụng CAD, thực tế ảo, mô phỏng khoa học, mô phỏng thông tin, phát triển trò chơi.

# Problem Statement

* OpenGL hỗ trợ những gì cho lập trình viên có thể xây dựng các ứng dụng đồ họa 3D?
* Cách sử OpenGL cho các tác vụ 3D (tĩnh và động) như thế nào?
* Xây dựng một ứng dụng 3D hữu ích dựng vào OpenGL và kiến thức về Đồ Họa Máy Tính.

# Technical Report

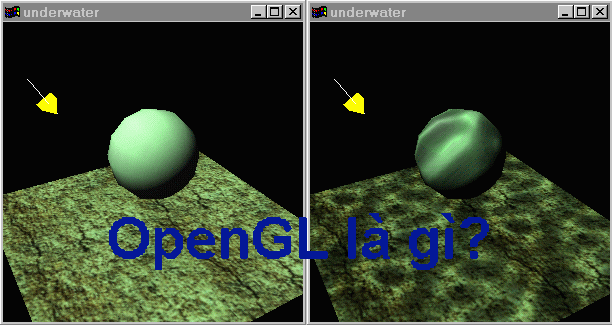
## Cách sử dụng OpenGL trong các tác vụ 3D (tĩnh và động)[1] [2] [7]

### Tổng quan về OpenGL

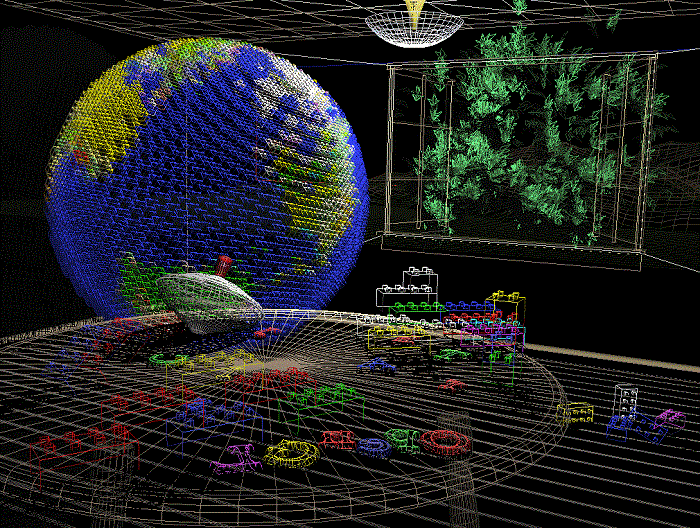
#### Open GL là gì? [6]

OpenGL là bộ thư viện đồ họa có khoảng 150 hàm giúp xây dựng các đối tượng và giao tác cần thiết trong các ứng dụng tương tác 3D.

OpenGL (Open Graphics Library) là giao diện chương trình ứng dụng chuẩn (API) của máy tính để định nghĩa các hình ảnh đồ họa 2 chiều và 3 chiều. Trước OpenGL, bất kỳ công ty nào phát triển một ứng dụng đồ họa thường phải viết lại phần đồ họa của nó cho từng nền tảng hệ điều hành và cũng phải biết đền phần cứng của đồ họa. Với OpenGL, một ứng dụng có thể tạo ra những hiệu ứng tương tự trong bất kỳ hệ điều hành nào bằng bất kỳ bộ tiếp hợp đồ họa OpenGL nào.



OpenGL chỉ định một tập hợp các lệnh hoặc thực hiện các chức năng. Mỗi lệnh chỉ đạo một hành động vẽ hoặc tạo ra các hiệu ứng đặc biệt. Một danh sách các lệnh này có thể được tao ra cho các hiệu ứng lặp đi lặp lại. OpenGL độc lập với các tính năng của các hệ điều hành Windows, nhưng cung cấp các tính năng “glue” cho các hệ điều hành cho phép một lượng lớn các chức năng tích hợp có thể yêu cầu thông qua API. Chúng bao gồm loại bỏ bề mặt bị ẩn, pha trộn alpha, chống răng cưa, lập bản đồ kết cấu, xem và biến đổi mô hình, các hiệu ứng khi quyển (sương mù, khói…)



OpenGL được phát triển bởi Silicon Graphics - là nhà phát triển workstation cao cấp. Các công ty khác tham gia bao gồm DEC, Intel, IBM, Microsoft và Sun Microsystems. Bạn sẽ phải học nếu muốn sử dụng OpenGL API. Microsoft cung cấp miễn phí tài tài liệu OpenGL cho các hệ thống Windows.

Bạn có thể coi OpenGL giống như ngôn ngữ đồ họa độc lập hoàn toàn và có thể tương thích với nhiều nền tảng máy tính khác nhau bao gồm các máy tính không có card đồ họa cao cấp.

**Công dụng của OpenGL:**

OpenGL có thể được sử dụng cho nhiều mục đích khác nhau, dưới đây là các tính năng chính mà OpenGL mang lại.

* Hỗ trợ tương tác các mô hình trong không gian 3 chiều thành một khối thống nhất đơn giản hơn.
* Ép buộc các phần cứng 3 chiều khác nhau phải tương thích với nhau thông qua các chức năng giao diện của OpenGL. Trường hợp hệ thống không thể ép phần cứng hỗ trợ hoàn toàn, OpenGL có thể sử dụng phần mềm để xử lý.
* Tạo ra các khối hình học có chiều sâu hơn.



**Ưu điểm của OpenGL:**

* **Đơn giản hóa việc phát triển phần mềm, tăng tốc các ứng dụng:**

OpenGL có thể đơn giản hóa việc phát triển phần mềm đồ họa, từ việc tạo ra một hình đa giác, đường kẻ hoặc đơn giản là tạo ra bề mặt cong có ánh sáng và kết cấu phức tạp nhất. OpenGL cho phép các nhà phát triển phần mềm truy cập vào các hình ảnh nguyên thủy, hiển thị danh sách, chuyển đổi mô hình, ánh sáng và texturing, chống răng cưa, trộn và nhiều tính năng khác.

Triển khai OpenGL phù hợp bao gồm đầy đủ các chức năng của OpenGL. Chuẩn OpenGL đã được xác định rõ ràng có các ràng buộc cho ngôn ngữ C, C++, Fortran, Ada và Java. Các ứng dụng sử dụng các chức năng của OpenGL có thể dễ dàng thay đổi sang các nền tảng khác nhau giúp tăng năng suất và giảm thời gian làm việc.

* **Có sẵn ở mọi nơi:** OpenGL được hỗ trợ trên tất cả các máy trạm UNIX và được sử dụng tiêu chuẩn trên mọi máy tính Windows, MacOS. Không có một API đồ họa nào hoạt động trên một phạm vi rộng hơn OpenGL.
* **Cấu trúc linh hoạt và nhiều khác biệt: Extensions**: Các nhà cung cấp nền tảng có quyền tự do thiết kế một ứng dụng OpenGL cụ thể để đáp ứng nhu cầu qua đó giảm chi phí, tăng hiệu suất hệ thống. Sư linh hoạt trong việc triển khai này có nghĩa là tăng tốc phần cứng OpenGL giúp cho ứng dụng có thể hiển thị đơn giản trên máy tính cấu hình thấp hoặc đầy đủ trên các máy tính cao cấp. Các nhà phát triển ứng dụng sẽ được đảm bảo kết quả hiển thị nhất quán bất chấp việc triển khai trên các nền tảng khác nhau.
* **Hỗ trợ các API nâng cao**: Các nhà phát triển phần mềm hàng đầu sử dụng OpenGL kết hợp với các thư viện rendering mạnh mẽ giúp tạo ra các nền tảng đồ họa 2D/3D cho các API cao cấp hơn. Nhà phát triển tận dụng các khả năng của OpenGL để cung cấp các giải pháp có sự khác biệt cao nhưng vẫn được hỗ trợ rộng rãi.
* **Luôn đổi mới**: OpenGL đang không ngừng phát triển. Các sửa đổi chỉnh thức diễn ra theo thời gian định kỳ và có nhiều mở rộng cho các nhà phát triển ứng dụng truy cập vào các thay đổi phần cứng mới nhất. Khi các Extensions được chấp nhận rộng rãi, chúng được xem xét để đưa vào tiêu chuẩn OpenGL lõi. Quá trình này cho phép OpenGL phát triển theo hướng sáng tạo nhưng vẫn được kiểm soát.

**Những thứ OpenGL không hỗ trợ:**

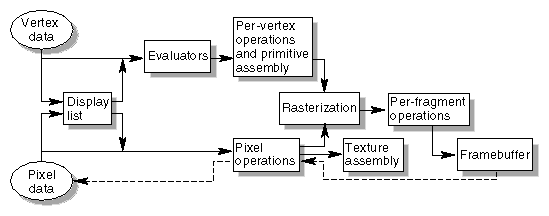
* Bản thân OpenGL không có sẵn các hàm nhập xuất hay thao tác trên window.
* OpenGL không có sẵn các hàm cấp cao để xây dựng các mô hình đối tượng, thay vào đó, người dùng phải tự xây dựng từ các thành phần hình học cơ bản ( điểm, đoạn thẳng, đa giác).

Rất may là một số thư viện cung cấp sẵn một số hàm cấp cao được xây dựng nên từ OpenGL. GLUT (OpenGL Utility Toolkit) là một trong số đó và được sử dụng rộng rãi.

**Những thứ OpenGL hỗ trợ là các hàm đồ họa:**

* Xây dựng các đối tượng phức tạp từ các thành phần hình học cơ bản (điểm, đoạn, đa giác, ảnh, bitmap).
* Sắp xếp đối tượng trong 3D và chọn điểm thuận lợi để quan sát.
* Tính toán màu sắc của các đối tượng (màu sắc của đối tượng được quy định bởi điều kiện chiếu sáng, texture của đối tượng, mô hình được xây dựng hoặc là kết hợp của cả 3 yếu tố đó).
* Biến đổi những mô tả toán học của đối tượng và thông tin màu sắc thành các pixel trên màn hình (quá trình này được gọi là resterization).

#### Mô hình hoạt động của OpenGL



OpenGL có cơ chết hoạt động kiểu ống dẫn tức là đầu ra của giai đoạn trước là đầu vào của giai đoạn sau. Từ sơ đồ thì các thành phần của cơ chế được giải thích như sau:

* Display List: Là nơi lưu lại một số lệnh để xử lý sau.
* Evaluator: Xấp xỉ các đường cong và mặt phẳng hình học bằng cách đánh giá các đa thức của dữ liệu đưa vào.
* Per-vertex operations and primitive assembly: Xử lý các primitive (điểm, đoạn, đa giác) được mô tả bởi các vertex. Các vertex sẽ được xử lý và các primitive được cắt xén vào viewport để chuẩn bị cho khâu kế tiếp.
* Rasterization: sinh ra một loạt các địa chỉ framebuffer và các giá trị liên quan bằng cách sử dụng mô tả 2 chiều của điểm, đoạn, đa giác. Mỗi phần tử (fragment) được sinh ra sẽ đưa vào giai đoạn kế tiếp.
* Per-fragment operations: Các tác vụ sau cùng (cập nhật có điều kiện cho framebuffer dựa vào dữ liệu vào và dữ liệu được lưu trữ trước đó của giá trị z (đối với z buffering), thực hiện trộn màu cho các pixel và làm một số thao tác khác) sẽ được thực hiện trên dữ liệu trước khi nó được chuyển thành pixel và đưa vào framebuffer.
* Trong trường hợp dữ liệu vào ở dạng pixel không phải vertex, nó sẽ đưa thẳng vào giai đoạn xử lý pixel. Sau giai đoạn này, dữ liệu ở dạng pixel sẽ được lưu vào texture memory để đưa vào giai đoạn Per-fragment operation hoặc đưa vào Rasterization như dữ liệu dạng Vertex (tức là các điểm).

#### Cấu trúc lệnh trong OpenGL

OpenGL sử dụng tiền tố gl và tiếp theo đó là những từ được viết hoa ở chữ cái đầu để tạo nên tên của một lệnh, ví dụ glClearColor(). Tương tự, OpenGL đặt tên các hằng số bắt đầu bằng GL\_ và các từ tiếp sau đều được viết hoa và cách nhau bởi dấu ‘\_’, ví dụ: GL\_COLOR\_BUFFER\_BIT.

Bên cạnh đó, với một số lệnh, để ám chỉ số lượng cũng như kiểu tham số được truyền, một số hậu tố được sử dụng như trong bảng sau:

|  |  |  |  |
| --- | --- | --- | --- |
| **Hậu tố** | **Kiểu dữ liệu** | **Tương ứng với kiểu trong C** | **OpenGL** |
| B | 8-bit integer | signed char | Glbyte |
| S | 16-bit integer | short | Glshort |
| I | 32-bit integer | int or long | GLint, Glsizei |
| F | 32-bit floating-point | float | GLfloat, Glclampf |
| D | 64-bit floating-point | double | GLdouble, GLclampd |
| Ub | 8-bit unsigned integer | unsigned char | GLubyte, GLboolean |
| Us | 16-bit unsigned integer | unsigned short | GLushort |
| Ui | 32-bit unsigned integer | unsigned int or unsigned long | GLuint, GLenum, GLbitfield |

**Ví dụ:** glVertex2i(1,3) tương ứng với xác định một điểm (x,y) với x, y nguyên (integer).  
**Lưu ý:** OpenGL có định nghĩa một số kiểu biến, việc sử dụng các định nghĩa này thay vì định nghĩa có sẵn của C sẽ tránh gây lỗi khi biên dịch code trên một hệ thống khác.  
Một vài lệnh của OpenGL kết thúc bởi v ám chỉ rằng tham số truyền vào là một vector.  
**Ví dụ:** glColor3fv(color\_array) thì color\_array là mảng 1 chiều có 3 phần tử là float.

#### OpenGL Utility Toolkit (GLUT)

Để khắc phục một số nhược điểm của OpenGL, GLUT được tạo ra với với nhiều hàm hỗ trợ:

* Quản lý window.
* Display callback.
* Nhập xuất (bàn phím, chuột,…).
* Vẽ một số đối tượng 3D phức tạp (mặt cầu, khối hộp,…).

Tên các hàm của GLUT đều có tiền tố là glut. Để hiểu rõ hơn về GLUT, người đọc tham khảo [ở đây](http://glprogramming.com/red/appendixd.html).

Để khai báo sử dụng OpenGL và GLUT, chúng ta download [ở đây](http://www.opengl.org/resources/libraries/glut/glut_downloads.php#windows), và chép các file sau vào trong cùng thư mục của project: glut.h, glut32.dll, glut32.lib.

### Vẽ các đối tượng hình học cơ bản

#### Một số thao tác cơ bản

**Xóa màn hình:**

Trong OpenGL có 2 loại buffer phổ biến nhất

* *color buffer*: buffer chứa màu của các pixel cần được thể hiện.
* *depth buffer* (hay còn gọi là z-buffer): buffer chứa chiều sâu của pixel, được đo bằng khoảng cách đến mắt. Mục đích chính của buffer này là loại bỏ phần đối tượng nằm sau đối tượng khác.

Mỗi lần vẽ, chúng ta nên xóa buffer:

|  |  |
| --- | --- |
| glClearColor(0.0, 0.0, 0.0, 0.0);  glClearDepth(1.0);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); | /\* xác định màu để xóa color buffer (màu đen) \*/  /\* xác định giá trị để xóa depth buffer \*/  /\* xóa color buffer và depth buffer \*/ |

**Xác định màu:** Khi vẽ một đối tượng, OpenGL sẽ tự động sử dụng màu đã được xác định trước đó. Do đó, để vẽ đối tượng với màu sắc theo ý mình, cần phải thiết lập lại màu vẽ. Thiết lập màu vẽ mới dùng hàm glColor3f(), ví dụ:

|  |  |
| --- | --- |
| glColor3f(0.0, 0.0, 0.0);  glColor3f(1.0, 0.0, 0.0);  glColor3f(0.0, 1.0, 0.0);  glColor3f(1.0, 1.0, 0.0);  glColor3f(0.0, 0.0, 1.0);  glColor3f(1.0, 0.0, 1.0);  glColor3f(0.0, 1.0, 1.0);  glColor3f(1.0, 1.0, 1.0); | // black // red // green // yellow // blue // magenta // cyan // white |

#### Vẽ các đối tượng hình học

***OpenGL không có sẵn các hàm để xây dựng các đối tượng hình học phức tạp, người dùng phải tự xây dựng chúng từ các đối tượng hình học cơ bản mà OpenGL hỗ trợ: điểm, đoạn thẳng, đa giác.***

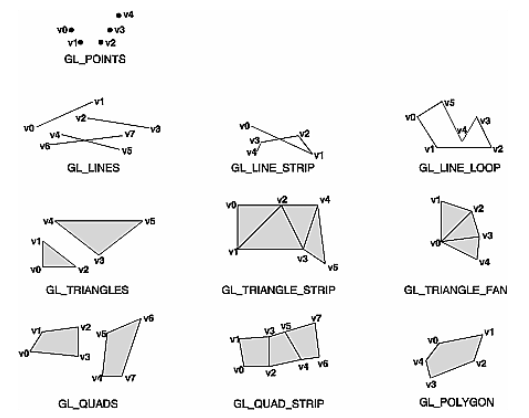
Khai báo một điểm, dùng hàm glVertexXY với X là số chiều (2, 3, hoặc 4), Y là kiểu dữ liệu.

Việc xây dựng các đối tượng hình học khác đều có thể được thực hiện như sau:

|  |
| --- |
| glBegin(mode); /\* xác định tọa độ và màu sắc của các điểm của hình \*/ glEnd(); |

mode có thể là một trong những giá trị sau:

|  |  |
| --- | --- |
| ***Giá trị*** | ***Ý nghĩa*** |
| GL\_POINTS | individual points |
| GL\_LINES | pairs of vertices interpreted as individual line segments |
| GL\_LINE\_STRIP | series of connected line segments |
| GL\_LINE\_LOOP | same as above, with a segment added between last and first vertices |
| GL\_TRIANGLES | triples of vertices interpreted as triangles |
| GL\_TRIANGLE\_STRIP | linked strip of triangles |
| GL\_TRIANGLE\_FAN | linked fan of triangles |
| GL\_QUADS | quadruples of vertices interpreted as four-sided polygons |
| GL\_QUAD\_STRIP | linked strip of quadrilaterals |
| GL\_POLYGON | boundary of a simple, convex polygon |

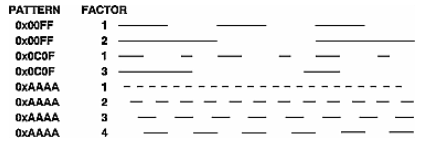


Màu sắc thôi chưa đủ, một số tính chất của điểm và đoạn cần quan tâm có thể được thiết lập qua các hàm:

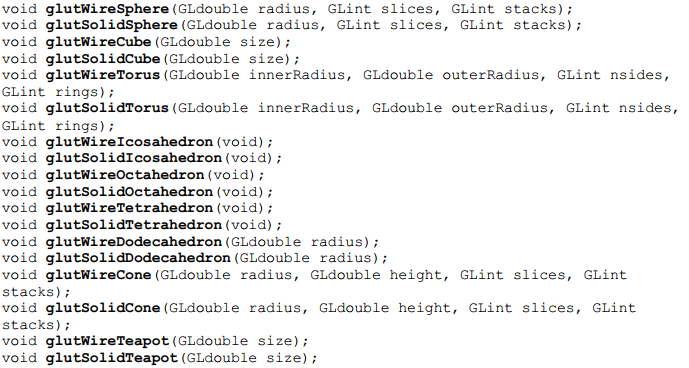
* Kích thước của một điểm: void **glPointSize**(GLfloat size).
* Độ rộng của đoạn thẳng: void **glLineWidth**(GLfloat *width*).
* Kiểu vẽ.

|  |
| --- |
| glEnable(GL\_LINE\_STIPPLE); // enable kiểu vẽ glLineStipple(factor, pattern); // pattern được cho trong bảng sau, factor thường là 1 /\* thực hiện các thao tác vẽ \*/ ... glDisable (GL\_LINE\_STIPPLE); // disable kiểu vẽ |

factor, pattern có thể là một trong những giá trị sau:



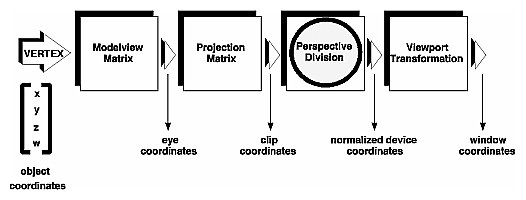
**GLUT có hỗ trợ sẵn một số hàm để vẽ các đối tượng hình học phức tạp hơn:**



### Viewing

#### Giới thiệu

Trong OpenGL, tiến trình đi từ điểm trong không gian thế giới thực đến pixel trên màn hình như sau:



**Tương ứng với các thao tác trong chụp ảnh như sau:**

|  |  |
| --- | --- |
| http://glprogramming.com/red/images/Image48.gif | Thiết lập chân máy và hướng máy ảnh về góc chụp.   Sắp xếp cảnh vật để chụp được ảnh như mong muốn.  Chọn ống kính cho máy ảnh hoặc điều chỉnh zoom.  Xác định xem bạn muốn bức ảnh cuối cùng lớn bao nhiêu.  ***ví dụ:*** ***bạn có thể muốn nó phóng to.***  Sau khi các bước này được thực hiện, hình ảnh có thể được chụp hoặc cảnh có thể được vẽ. |

Trong OpenGL các điểm được biểu diễn dưới hệ tọa độ thuần nhất. Do đó, tọa độ của một điểm 3D được thể hiện bởi *(x,y,z,w)T*, thông thường w = 1. Một phép biến đổi trên một điểm v tương ứng với việc nhân v với ma trận biến đổi M kích thước 4x4: *v’ = M.v*.

Trong mỗi bước ModelView và Projection (chiếu), tại mỗi thời điểm, OpenGL đều lưu trữ một ma trận biến đổi hiện hành. Để thông báo với chương trình rằng sẽ thực thi bước ModelView, chúng ta cần phải gọi hàm

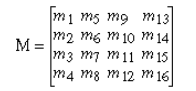
glMatrixMode(GL\_MODELVIEW)  
Tương tự, để thông báo cho bước Projection, chúng ta gọi hàm

glMatrixMode(GL\_PROJECTION)

Để thiết lập ma trận biến đổi hiện hành bằng ma trận M, chúng ta dùng hàm sau

void **glLoadMatrix**{fd}(const TYPE \*m);

***Chú ý*:** ma trận M có dạng



Vì một lí do nào đó chúng ta phải thay đổi ma trận hiện hành, nhưng sau đó chúng ta lại muốn khôi phục lại nó. Ví dụ như chúng ta dời tới một điểm nào đó để vẽ khối hộp, sau đó chúng ta muốn trở lại vị trí ban đầu. Để hỗ trợ các thao tác lưu trữ ma trận hiện hành, OpenGL có một stack cho mỗi loại ma trận hiện hành, với các hàm sau:

* Đẩy ma trận hiện hành vào trong stack: void **glPushMatrix**(void)
* Lấy ma trận hiện hành ở đỉnh stack: void **glPopMatrix**(void)

#### Thao tác trên ModelView

Trước khi thực hiện các thao tác trên ModelView, chúng ta cần gọi hàm

glMatrixMode(GL\_MODELVIEW);

##### Các phép biến đổi affine

OpenGL hỗ trợ sẵn các hàm biến đổi affine cơ bản như sau:

* Tịnh tiến

void **glTranslate**{fd}(TYPE x, TYPE y, TYPE z);

* Quay quanh trục nối gốc tọa độ với điểm (x,y,z)

void **glRotate**{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

* Tỉ lệ (tâm tỉ lệ tại gốc tọa độ)

void **glScale**{fd}(TYPE x, TYPE y, TYPE z);

Với mục đích tổng quát hơn, việc nhân ma trận M có thể được thực thi bởi hàm: void **glMultMatrix**{fd}(const TYPE \*m);

**Chú ý:**

* Mọi thao tác biến đổi trên đều có nghĩa là lấy ma trận biến đổi hiện hành nhân với ma trận biến đổi affine cần thực hiện.
* Thứ tự thực hiện sẽ ***ngược*** với suy nghĩ của chúng ta, ví dụ thứ tự thực hiện mà chúng ta nghĩ là: quay quanh trục z một góc , sau đó tịnh tiến đi một đoạn (trx, try, trz) thì sẽ được thực thi trong OpenGL như sau:

glTranslatef(trx, try, trz)

glRotatef(, 0, 0, 1)

##### Thiết lập view

Giống như chụp hình, thiết lập view là thiết lập vị trí cũng như góc, hướng của camera. GLUT có một hàm giúp thiết lập view một cách nhanh chóng:

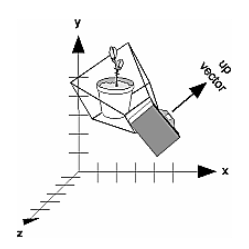
*void* ***gluLookAt****(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);*

**Trong đó:**

* (eyex, eyey, eyez) là vị trí đặt của view.
* (centerx, centery, centerz) là điểm nằm trên đường thẳng xuất phát từ tâm view hướng ra ngoài.
* (upx, upy, upz) là vector chỉ hướng lên trên của view.

***Ví dụ:***

* (eyex, eyey, eyez) = (4, 2, 1)
* (centerx, centery, centerz) = (2, 4, -3)
* (upx, upy, upz) = (2, 2, -1)



#### Thao tác trên Projection (Phép chiếu)

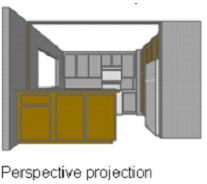
Trước khi thực hiện các thao tác chiếu, chúng ta gọi 2 hàm:

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

##### Chiếu phối cảnh (Perspective Projection)

Đặc điểm của phép chiếu này là đối tượng càng lùi ra xa thì trông càng nhỏ.



Để thiết lập phép chiếu này, OpenGL có hàm sau:

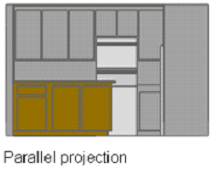
|  |
| --- |
| void **glFrustum**(GLdouble left, GLdouble right, GLdouble bottom,Gldouble top, GLdouble near, GLdouble far); |
| chap3-5.gif |

**Ngoài ra, để dễ dàng hơn, chúng ta có thể sử dụng hàm:**

|  |
| --- |
| void **gluPerspective**(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far); |
|  |

##### Chiếu trực giao (Orthogonal Projection)

Trong phép chiếu này, khoảng cách của vật tới camera không ảnh hưởng tới độ lớn của vật đó khi hiển thị.



|  |
| --- |
| void **glOrtho**(GLdouble left, GLdouble right, GLdouble bottom, Gldouble top, GLdouble near, GLdouble far); |
| http://glprogramming.com/red/images/Image63.gif |

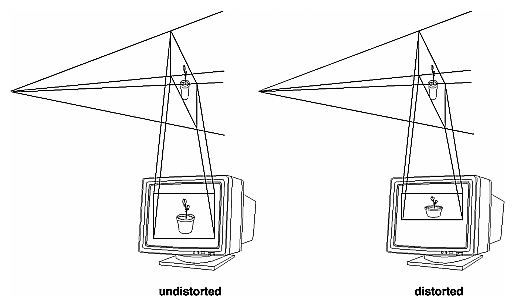
#### Thao tác trên Viewport

OpenGL có hàm để thiết lập viewport:

void **glViewport**(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

***Trong đó*** (x,y) là vị trí điểm trái-trên trong cửa sổ vẽ, width, height là chiều rộng và cao của viewport. Mặc định (x,y,width,height) = (0,0,winWidth, winHeight) (chiếm toàn bộ cửa sổ)

Hình sau minh họa việc thiết lập viewport:



**Chú ý:**

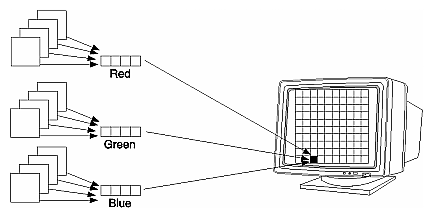
|  |  |
| --- | --- |
| Lập trình trong môi trường Windows (ví dụ như dùng MFC), tọa độ trong cửa sổ thông thường được quy định như hình bên. |  |
| Tuy nhiên, trong viewport, chúng ta cần phải quên quy ước đó đi, thay bằng hình bên. |  |
| **Khi bắt sự kiện mouse thì tọa độ trả về vẫn tuân theo quy tắc của Windows.** | |

### Chế độ màu

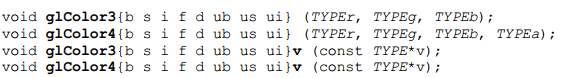
#### Chế độ màu RGBA

OpenGL hỗ trợ 2 chế độ màu: RGBA và Color-Index, chúng ta chỉ quan tâm đến RGBA.

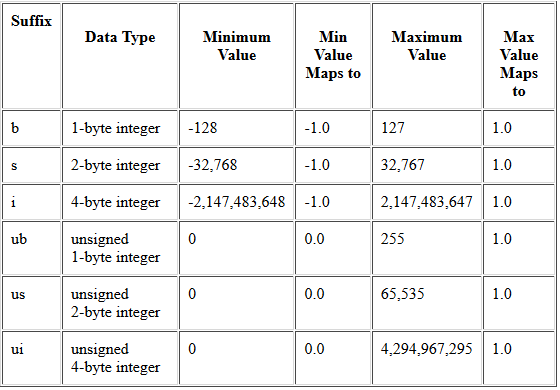
Trong chế độ màu RGBA, RGB lần lượt thể hiện màu Red, Green, Blue. Còn thành phần A (tức alpha) không thực sự ảnh hưởng trực tiếp lên màu pixel, người ta có thể dùng thành phần A để xác định độ trong suốt hay thông số nào đó cần quan tâm.



Để thiết lập màu vẽ hiện hành trong chế độ RGBA, chúng ta có thể sử dụng các hàm sau:



trong đó, nếu các tham số là số thực thì thành phần màu tương ứng sẽ nằm trong đoạn [0,1], ngược lại thì sẽ được chuyển đổi như ở bảng sau:



#### Thiết lập mô hình shading

Một đoạn thẳng có thể được tô bởi một màu đồng nhất (chế độ *flat*) hay bởi nhiều màu sắc khác nhau (chế độ *smooth*). Để thiết lập chế độ shading phù hợp, chúng ta có thể sử dụng hàm:

void **glShadeModel** (GLenum mode);

trong đó mode là chế độ mong muốn, nhận 1 trong 2 giá trị GL\_SMOOTH hoặc GL\_FLAT.

**Chế độ smooth:**

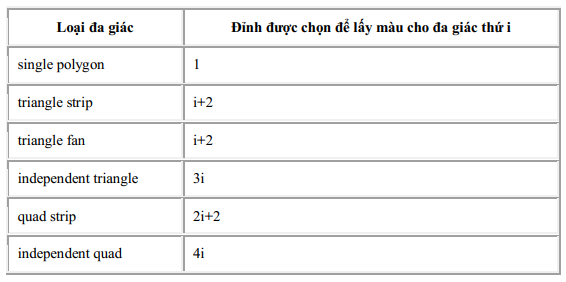
Ví dụ sau giúp chúng ta sẽ hiểu được chế độ smooth có tác động như thế nào?

|  |  |
| --- | --- |
| Code |  |
| Ouput |  |

**Chế độ flat:**

Chế độ flat tô hình đang xét một màu đồng nhất. Khi đó, OpenGL sẽ lấy màu của một đỉnh làm màu tô cho toàn bộ hình.

* Đối với đoạn thẳng, điểm đó là điểm cuối của đoạn.
* Đối với đa giác, điểm đó được chọn theo quy tắc trong bảng sau:



***Tuy nhiên, cách tốt nhất để tránh lầm lẫn là thiết lập màu tô đúng 1 lần.***

### Chiếu sáng

#### Các loại nguồn sáng

OpenGL có bốn loại nguồn sáng:

* Ánh sáng môi trường: đến từ mọi hướng cùng lúc; Ví dụ ánh sáng trong một phòng được chiếu sáng đầy đủ là ánh sáng môi trường.
* Ánh sáng khuếch tán: ánh sáng đến từ một hướng, chiếu vào bề mặt đối tượng, làm cho bề mặt này trở nên sáng chối hơn, sau đó ánh sáng bị khuếch tán đi mọi hướng.
* Ánh sáng phản chiếu: ánh sáng tạo đốm phản chói, thường là màu trắng, trên các bề mặt có tính phản chiếu cao.
* Nguồn phát là nguồn ánh sáng phát ra từ đối tượng như bóng đèn chẳng hạn.

#### Tạo nguồn sáng

OpenGL Cho phép có ***8 nguồn sáng*** khác nhau trong một chương trình. mỗi nguồn sáng có nhiều thuộc tính để kiểm soát các ánh sáng tác động đến một cảnh. Các ánh sáng này bao gồm ánh sáng môi trường, ánh sáng khuếch tán, ánh sáng phản chiếu và ánh sáng vị trí.

*void* ***glLight****{if}(GLenum light, GLenum pname, TYPEparam);  
void* ***glLight****{if}****v****(GLenum light, GLenum pname, TYPE \*param);*

trong đó:

* light chỉ nhận các giá trị: *GL\_LIGHT0, GL\_LIGHT1, ... hoặc GL\_LIGHT7* (do có tối đa 8 nguồn sáng).
* pname: đặc tính của ánh sáng

|  |  |  |
| --- | --- | --- |
| Parameter Name | Giá trị mặc định | Ý nghĩa |
| Gl\_AMBIENT | (0.0, 0.0, 0.0, 1.0) | Xác định ánh ánh sáng môi trường |
| GL\_CONSTANT\_ATTENUATION | 1.0 | Xác định lượng giảm ánh sáng theo một hệ số |
| GL\_DIFFUSE | (1.0, 1.0, 1.0, 1.0) | Xác định ánh sáng khuếch tán |
| GL\_LINEAR\_ATTENUATION | 0.0 | Xác định lượng giảm ánh sáng trên cơ sở khoảng cách từ nguồn sáng đến đối tượng |
| GL\_POSITION | (0.0, 0.0, 1.0, 0.0) | Xác định vị trí nguồn sáng |
| GL\_QUADRATIC\_ATTENUATION | 0.0 | Xác định lượng giảm ánh sáng theo bình phương khoảng cách từ nguồn sáng đến đố tượng |
| GL\_SPECULAR | (1.0, 1.0, 1.0, 1.0) | Xác định ánh sáng khoảng chiếu |
| GL\_SPOT\_CUTOFF | 180.0 | Xác định góc trải của đèn chiếu |
| GL\_SPOT\_DIRECTION | (0.0, 0.0, -1.0) | Xác định hướng đèn chiếu |
| GL\_SPOT\_EXPONENT | 0.0 | Xác định cường độ đèn chiếu ,có tính đến góc của ánh sáng |

* param: chỉ ra các giá trị mà đặc tính pname được đặt; đó là một con trỏ đến một nhóm các giá trị nếu phiên bản vectơ được sử dụng, hoặc bản thân giá trị nếu phiên bản nonvector được sử dụng. Phiên bản nonvector có thể được sử dụng để chỉ thiết lập các đặc tính ánh sáng có giá trị đơn.

**Ví dụ:**

GLfloat light\_ambient[] = { 0.0, 0.0, 0.0, 1.0 };

GLfloat light\_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat light\_specular[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat light\_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse);

glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular);

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);

Khi đã có định nghĩa nguồn sáng, ta phải kích hoạt chúng, tương tự như việc bậc công tắt đèn, bằng hàm glEnable():

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);

Lời gọi đầu kích hoạt chiếu sáng. Lời gọi thứ hai bật nguồn sáng 0.

#### Tính chất vật liệu

Màu của đối tượng trong cảnh phụ thuộc vào màu của ánh sáng mà nó phản xạ . Ví dụ, nếu đối tượng phản xạ ánh sáng xanh lục, thì nó sẽ có màu xanh lục trong ánh sáng trắng; Do các thành phần màu đỏ và xanh dương của ánh sáng bị hấp thụ chỉ còn lại thành phần xanh lục đến được mắt người.

Thông thường, cả ánh sáng môi trường và ánh sáng khuếch tán phản xạ từ đối tượng theo cùng một cách. Tức là nếu đối tượng phản xạ ánh sáng xanh lục, thì nó phản xạ màu xanh lục đối với ánh sáng môi trường lẫn ánh sáng khuếch tán. Trong khi ánh sáng phản chiếu hầu như luôn luôn cùng màu với ánh sáng chiếu vào đối tượng . Ví dụ, xét một khối vuông được thể hiện trong ánh sáng vàng (màu vàng là tổ hợp đồng lượng của màu đỏ và xanh dương). Nếu muốn khối vuông có màu đỏ, vật liệu của nó phải phản xạ màu đỏ của ánh sáng mội trường và ánh sáng khuếch tán. Để làm khối vuông trở nên sáng hơn, thì phải thiết lập cho nó tính phản xạ màu vàng đối với ánh sáng phản chiếu.

OpenGL cho phép định nghĩa các tính chất của đối tượng, để xác định loại ánh sáng mà đối tượng phản xạ, và như vậy gián tiếp xác định màu cuối cùng của đối tượng.

**Thiết lập các đặt tính vật liệu:**

void **glMaterial**{if}(GLenum face, GLenum pname, TYPEparam);

void **glMaterial**{if}**v**(GLenum face, GLenum pname, TYPE \*param);

trong đó:

* face chỉ nhận các giá trị: GL\_FRONT, GL\_BACK, hoặc GL\_FRONT\_AND\_BACK (cho biết mặt đối tượng được áp dụng).
* pname: thuộc tính vật liệu

|  |  |  |
| --- | --- | --- |
| Parameter Name | Giá trị mặc định | Ý nghĩa |
| GL\_AMBIENT | (0.2, 0.2, 0.2, 1.0) | Ánh sáng môi trường bị ánh xạ. |
| GL\_AMBIENT\_AND\_DIFFUSE |  | Ánh sáng môi trường và khuếch tán bị ánh xạ. |
| GL\_COLOR\_INDEXES | (0,1,1) | Xác định các chỉ số màu cho ánh sáng. |
| GL\_DIFFUSE | (0.8, 0.8, 0.8, 1.0) | Ánh sáng khuếch tán bị phản xạ. |
| GL\_EMISSION | (0.0, 0.0, 0.0, 1.0) | Xác định nguồn sánh phát. |
| GL\_SHININESS | 0.0 | Xác định độ bóng vật liệu. |
| GL\_SPECULAR | (0.0, 0.0, 0.0, 1.0) | Ánh sáng phản chiếu bị phản xạ. |

* param: chỉ ra các giá trị mà đặc tính pname được đặt; đó là một con trỏ tới một nhóm giá trị (nếu phiên bản vectơ được sử dụng) hoặc giá trị thực (nếu phiên bản nonvector là đã sử dụng).

***Ví dụ***:

Glfloat materialAmbient[] = {0.0f, 0.7f, 0.0f, 1.0f};

Glfloat materialSpecular[] = {1.0f, 1.0f, 1.0f, 1.0f};

GlMaterialfv (GL\_FRONT, GL\_AMBIENT\_AND\_DIFFUSE, materialAmbient);

GlMaterialfv (GL\_FRONT, GL\_SPECULAR, materialSpecular);

Mỗi mảng trên chứa các giá trị RGBA cho một kiểu phản xạ ánh sáng riêng biệt, đó là các giá trị RGBA tương ứng với màu sắc ánh sáng dội ra từ vật liệu đối tượng.

### Pha trộn, giảm hiệu ứng răng cưa, sương mù

#### Pha trộn

Pha trộn (blending) là phương pháp kết hợp màu của các pixel nguồn và đích theo các cách khác nhau để tạo những hiệu quả đặt biệt. Pha trộn thường được sử dụng để tạo các đối tượng trong mờ. Khi một đối tượng được pha trộn chồng lên đối tượng khác thì có thể nhìn xuyên qua đối tượng được bao phủ do màu của đối tượng nguồn kết hợp với màu của pixel được bao phủ tạo ra màu mới.

**Cách sử dụng pha trộn:**

* Kích hoạt

glEnable(GL\_BLEND); // GL\_BLEND báo OpenGL cho phép pha trộn.

glDisable(GL\_BLEND);// tắt pha trộn

* Lựa chọn chế độ pha trộn

*void* ***glBlendFunc****(GLenum sfactor, GLenum dfactor);*

trong đó, sfactor và dfactor lần lượt xác định các hàm pha trộn nguồn và đích, nhận các giá trị như bảng bên dưới:

|  |  |  |
| --- | --- | --- |
|  | **Hằng số** | **Hệ số kết quả** |
| sfactor | GL\_DST\_ALPHA | (AD , AD , AD , AD ) |
| GL\_DST\_COLOR | (RD , GD , BD ,AD ) |
| GL\_ONE | (1,1,1,1) |
| GL\_ONE\_MINUS\_ DST\_ALPHA | (1,1,1,1) - (AD , AD , AD , AD ) |
| GL\_ONE\_MINUS\_ DST\_COLOR | (1,1,1,1) - (RD , GD , BD ,AD ) |
| GL\_ONE\_MINUS\_SRC\_ALPHA | (1,1,1,1) - (AS,AS,AS,AS) |
| GL\_ SRC\_ALPHA | (AS,AS,AS,AS) |
| GL\_ONE\_MINUS\_SRC\_COLOR | (F,F,F,1) với F= min( AS,(1-A0) ) |
| GL\_ZERO | ( 0,0,0,0 ) |
| dfactor | GL\_DST\_ALPHA | (AD , AD , AD , AD ) |
| GL\_ONE | (1,1,1,1) |
| GL\_ONE\_MINUS\_ DST\_ALPHA | (1,1,1,1) - (AD , AD , AD , AD ) |
| GL\_ONE\_MINUS\_SRC\_ALPHA | (1,1,1,1) - (AS,AS,AS,AS) |
| GL\_ONE\_MINUS\_SRC\_COLOR | (1,1,1,1) - (RS , GS , BS , AS ) |
| GL\_ SRC\_ALPHA | (AS,AS,AS,AS) |
| GL\_ SRC\_COLOR | (RS , GS , BS , AS ) |
| GL\_ZERO | ( 0,0,0,0 ) |

Ý nghĩa các bảng này như sau: Để quyết định màu sắc cho một pixel được pha trộn, OpenGL phải nhân các thành phần R,G,B,A của màu nguồn và đích với một hệ số. Hệ số này được quyết định bởi các hàm pha trộn nguồn và đích . Mọi tính toán được thể hiện trong các bảng trên dẫn đến bốn giá trị mà OpenGL dùng để nhân với các thành phần màu R, G, B, A.

**Ví dụ:**

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

GL\_SRC\_ALPHA có nghĩa là OpenGL nhân các thành phần màu nguồn với gia trị alpha của màu nguồn.

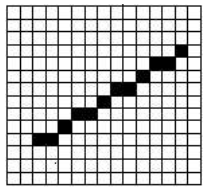
Với GL\_ONE\_MINUS\_SRC\_ALPHA thì trước hết giá trị alpha của màu nguồn phải được trừ đi 1, sau đó nhân kết quả phép trừ với các giá trị RGBA đích.

Giả sử pixel nguồn (là pixel mà OpenGL sắp sửa vẽ ) có các gía trị  
RGBA (0.7 ,0.5 ,0.8 ,0.6 ) và pixel đích ( là pixel mà OpenGL sẽ vẽ pixel nguồn lên) có các giá trị RGBA (0.6, 0.3, 0.6, 1.0 ). Trước tiên OpenGL cung cấp hàm  
GL\_SRC\_ALPHA cho các giá trị RGBA nguồn, tức là nhân 0.6 cho 0.7, 0.5 , 0.8, và 0.6.

Kết quả nhận được các giá trị pixel nguồn là (0.42, 0.3, 0.48 , 0.36). Sau đó OpenGL cung cấp hàm GL\_ONE\_MINUS\_SRC\_ALPHA cho pixel đích, tức là lấy 1 trừ đi giá trị alpha nguồn, được 0.4 rồi nhận kết quả này cho 0.6, 0.3, 0.6 và 1.0  
Kết quả nhận được các giá trị pixel đích là (0.24, 0.12, 0.24, 0.4 )

Như vậy với chọn lựa GL\_SRC\_ALPHA là hàm pha trộn nguồn và  
GL\_ONE\_MINUS\_SRC\_ALPHA là hàm pha trộn đích, giá trị alpha nguồn sẽ quyết định phần trăm mỗi màu được OpenGL kết hợp để tạo nên màu cuối cùng. Với giá trị alpha 0.6, 60% màu đến từ nguồn và 40% màu đến từ đích Giá trị alpha càng lớn thì kết quả càng trở nên mờ đục. Do đó có thể coi giá trị alpha như độ mờ đục muốn vẽ.

#### Giảm hiệu ứng răng cưa

* Trên màn hình một đường thẳng là tập hợp của các pixel được chiếu sáng trong hệ thống kẽ ô vuông.
* Do đó chỉ có đường nằm ngang hay thẳng đứng là được vẽ một cách suông sẽ còn các đường nghiêng sẽ có hiện tượng răng cưa (alias).
* Độ phân giải càng cao thì hiện tượng răng cưa càng giảm nhưng không thể không có.

Giải pháp kĩ thuật để giảm hiệu ứng răng cưa (antialiasing ) là ***sử dụng các sắc thái màu khác nhau để che dấu cạnh răng cưa của đường thẳng trên màn hình***. Thực tế antialiasing chỉ là một kiểu hiệu quả pha trộn OpenGL.

**Cách sử dụng khử răng cưa**

* Kích hoạt Antialiasing

glEnable (GL\_LINE\_SMOOTH );

GL\_LINE\_SMOOTH báo cho OpenGL thực hiên antialiasing cho đường thẳng.

Đối với điểm hay đa giác, antialiasing được kích hoạt bởi các hằng GL\_POINT\_SMOOTH hay GL\_POLYGON\_

* Kích hoạt pha trộn cho Antialiasing

Do antialiasing là một hiệu quả pha trộn nên cần phải kích hoạt pha trộn và chọn hàm pha trộn :

glEnable (GL\_BLEND);

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

Với các hàm pha trộn khác nhau sẽ nhận được các kết quả antialiasing khác nhau. Nhưng thông dụng nhất là sử dụng hàm nguồn GL\_SRC\_ALPHA và hàm đích GL\_ONE\_MINUS\_SRC\_ALPHA.Cần nhớ rằng với chọn lựa như vậy, thành phần alpha của đối tượng sẽ ảnh hưởng tới kết quả antialiasing.

Ngoài ra các màu trong bảng màu logic sẽ quyết định độ chính xác của antialiasing.

* Hàm glHintt()

*void* ***glHint****(GLenum* target*, GLenum* hint*);*

trong đó, target giúp xác định thao tác đề nghị, có thể nhận các giá trị là

GL\_FOG\_HINT , GL\_LINE\_SMOOTH, GL\_PERSPECTIVE\_CORRECTION\_HINT, GL\_POINT\_SMOOTH\_HINT hay GL\_POLYGON\_SMOOTH\_HINT

hint là lời gợi ý , có thể là:

GL\_DONT CARE (cho phép OpenGL thực hiện tùy ý)

GLFASTEST (OpenGL thự hiện phương thức nhanh nhất) hay

GL\_NICEST (OpenGL thực hiện phương thức chính xác nhất).

**Vẽ đường thẳng không răng cưa:**

|  |
| --- |
| glClearColor (1.0f , 1.0f , 1.0f , 0.0f ); glClear (GL\_COLOR\_BUFFER\_BIT); glShadeModel (GL\_FLAT); glEnable (GL\_LINE\_SMOOTH); glEnable (GL\_BLEND); glBlendFunc(GL\_SRC\_ALPHA,GL\_ONE\_MINUS\_SRC\_ALPHA); glHint (GL\_LINE\_SMOOTH\_HINT,GL\_DONT\_CARE); glLineWidth (2.0); glColor4f (0.0f , 0.0f , 0.0f , 1.0f ); glauxWireSphere (1.0); glDisable (GL\_LINE\_SMOOTH); glDisable (GL\_BLEND); |

Trong trường hợp các hàm pha trộn nguồn, đích là GL\_SRC\_ALPHA và  
GL\_ONE\_MINUS\_SRC\_ALPHA , chọn lựa màu trên cho giá trị alpha bằng 1.0 , đường được vẽ đậm nhất. Với alpha bằng 0.0 sẽ không có đường nào được vẽ cả.

***Kết quả ảnh:***

|  |  |
| --- | --- |
| Hình mạng nhện  ***không dùng antialiasing*** | Hình mạng nhện  ***có dùng antialiasing*** |
|  |  |

#### Sương mù

Không khí cũng như mọi vật chất khác không hoàn toàn trong suốt nên ánh sáng đi qua nó sẽ bị hấp thụ một phần hay toàn bộ. Nói một cách khác, đối tượng càng ở xa thì thể hiện của nó càng mờ nhạt. OpenGL có thể xem sương mù như một loại kính lọc có hiệu quả càng lớn trên đối tượng càng xa. Bằng cách chọn lựa các hàm sương mù, màu sắc, mật độ v.v... ta có thể quyết định mức độ hiệu quả.

**Cách sử dụng sương mù**

* Kích hoạt

glEnable (GL\_FOG); //Hằng GL\_FOG báo cho OpenGL kích hoạt hiệu ứng sương mù

* Chọn hàm sương mù

glFogi (GL\_FOG\_MODE , GL\_LINE);

Đối số đầu tiên của hàm xác định thông số sương mù muốn thiết lập, có thể là

GL\_FOG\_DENSITY, GL\_FOG\_END, GL\_FOG\_INDEX , GL\_FOG\_MODE hay GL\_FOG\_START.

Đối số thứ hai là giá trị thiết lập cho thông số đã xác định bởi đối số thứ nhất.

Với GL\_FOG\_MODE, các giá trị có thể dùng ở đối số thứ hai là GL\_EXP,  
GL\_EXP2, GL\_LINEAR (glFog() có bốn phiên bản). GL\_EXP và GL\_EXP2 sử dụng các hàm số mũ để xác định màu của pixel sương mù. Tác  
dụng của GL\_EXP2 lớn hơn tác dụng của GL\_EXP. Do OpenGL quản lý hầu hết các chi tiết, các hàm GL\_EXP và GL\_EXP2 dễ sử dụng trong chương trình. Nhưng GL\_LINEAR hàm cung cấp hiệu ứng sương mù trên cơ sở kích thước vùng sương mù cho phép người dùng kiểm soát nhiều hơn. Với GL\_LINEAR có thể báo cho OpenGL chính xác nơi bắt đầu và kết thúc sương mù trong cảnh bằng lời gọi glFogf() :

glFogf (GL\_FOG\_START , 0.0f);

glFogf (GL\_FOG\_END, 10.0f );

Hằng GL\_FOG\_START báo cho OpenGL thiết lập điểm bắt đầu sương mù. Đối số thứ hai là khoảng cách kể từ điểm nhìn mà hiệu ứng fog bắt đầu. Hằng GL\_FOG\_END báo cho OpenGL thết lập điểm kết thúc sương mù, với đối số thứ hai cũng là khoảng cách kể từ điểm nhìn. Chỉ có thể thiết lập điểm bắt đầu và kết thúc sương mù khi dùng GL\_LINEAR.

Việc thiết lập là không hiệu lực khi dùng GL\_EXP và GL\_EXP2 . Tuy nhiên khi dùng GL\_EXP và GL\_EXP2 ta có thể thiết lập mật độ sương mù bằng lời gọi glFog() với GL\_FOG\_DENSITY làm đối số thứ nhất và giá trị mật độ làm đối số thứ hai. Khoảng cách giữa điểm bắt đầu và kết thúc sương mù càng lón thì hiệu quả sương mù càng kém. ***Ví dụ*** với điểm bắt đầu là 0.0 nếu điểm kết thúc là 5.0 thì các đối tượng trong vùng này sẽ chịu tác dụng sương mù nhiều, nhưng nếu điểm kết thúc là 20.0 thì hiệu quả sương mù bị giảm rõ rệt do tác dụng bị trải ra trên khoảng cách lớn hơn.

* Thiết lập màu sương mù

Với việc thiết lập màu sương mù có thể đạt được hiệu quả lớn hơn màu sắc sử dụng trong cảnh. Trong đa số trường hợp thường dùng sương mù màu trắng với độ sáng vừa phải:

GLfloat fogColor[]={0.6f , 0.6f , 0.6f , 1.0f};

GlFogfv (GL\_FOG\_COLOR , fogColor);

Hai đối số của hàm là thông số sương mù muốn thiết lập và địa chỉ mảng chứa giá trị thiết lập cho thông số. Có thể có được các kết quả thú vị bằng cách sử dụng các màu khác màu trắng. Sương mù màu đen cho phép tạo cảnh ban đêm hay trời tối.

* Thiết lập màu sương mù

Trong Antialiasing (giảm răng cưa), ta đã làm quen với glHint(). Hàm này cũng có thể sử dụng trong hiệu ứng sương mù vớiGL\_FOG\_HINT làm đối số thứ nhất:

glHint (GL\_FOG\_HINT , GL\_DONT\_CARE);

Cũng như phần Antialiasing GL\_DONT\_CARE cho phép OpenGL sử dụng tùy ý các phương pháp sương mù.

### Display Lists

#### Giới thiệu

Display Lists là một nhóm lệnh OpenGL được lưu lại sau khi thực thi. Được sử dụng khi ta có ý định vẽ lại hoặc thay đổi trạng thái nhiều lần một đối tượng, thi việc sử dụng Display List để làm tăng khả năng thực thi chương trình (trong mô hình client – server : display list sẽ làm giảm sự hao phí thời gian truyền dữ liệu).

**Tính chất:**

* Nếu một display list tạo ra thì không thể nào sửa đổi. Display list cũng làm việc tốt với các lệnh của thư viện gl. Có nhiều cách để thực hiện display list.
* Sự thực thi của display list không chậm hơn thực thi các lệnh được chứa bên trong nó một cách độc lập.

**Các trường hợp có thể sử dụng:**

* Các tác vụ trên ma trận.
* Raster các bitmaps và các ảnh.
* Đặc tính nguồn sáng, chất liệu và mô hình chiếu sáng.
* Textures.
* Polygon stiple pattern.

**Nhược điểm:**

* Tính không biến đổi của dislay list.
* Tốn vùng nhớ nếu cần lưu trữ dữ liệu được phân chia từ dislay list.
* Nếu danh sách là nhỏ thì không hiệu quả.

#### Thao tác với Display List

**Đặt tên vào Display List:**

* Để bắt đầu và kết thúc một dislay list, ta sử dụng 2 hàm:

glNewList (Gluint list, Glenum mode) ;

glEndList (void);

trong đó: list là tên của một dislay list, mỗi dislay list được đặt tên bằng một giá trị integer.

mode: GL\_COMPLE and GL\_COMPLE\_AND\_EXCUTE.

Chỉ có một dislay list tại một thời điểm, nên tránh sự trùng tên ta dùng hàm glGenList (Glsizei range).

* Khi không sử dụng dislay list ta dùng hàm:

GlDeleteList (Glint list, Glsizei range ); // Để xóa dislay list đã định nghĩa.

Hàm glIsList (Gluint list);// Dùng để kiểm tra dislay list có định nghĩa chưa.

**Những lệnh không được chứa trong Display List:**

glColorPointer();  
glFlush();  
glNormalPointer();  
glDeleteList();  
glGenList();  
glPixelStore();  
glDisableClienState();  
glGet\*();  
glReadPixel();  
glEdgeflagPointer ();

…..

**Để thực thi Display List:** Dislay list được thực thi bằng glCallList (Gluin list), có thể gọi glCallList (Gluinlist) ở bất cứ nơi nào trong chương trình miễn là OpenGL context truy cập dislay list tích cực.

**Cấp bật trong Dislay List:**

* Đó là dislay list mà nó được định nghĩa trong một dislay list khác. Để tránh sự đệ qui vô hạn chế số phần tử trong dislay list.
* OpenGL cho phép tạo một dislay list mà nó gọi tới một cái khác nhưng cái này chưa được nghĩa, kết quả là không có gì xảy ra.

**Quản Lý Biến Trạng Thái Trong Dislay List:**

* Các biến trạng thái của OpenGL sau khi ra khỏi dislay list vẫn còn ảnh hưởng đến các lệnh tiếp theo trong chương trình.
* Không thể dùng glGet\*() trong dislay list, do đó để lưu trữ và phục hồi biến trạng thái phải dùng glPushAttr() và glPopAttrib().

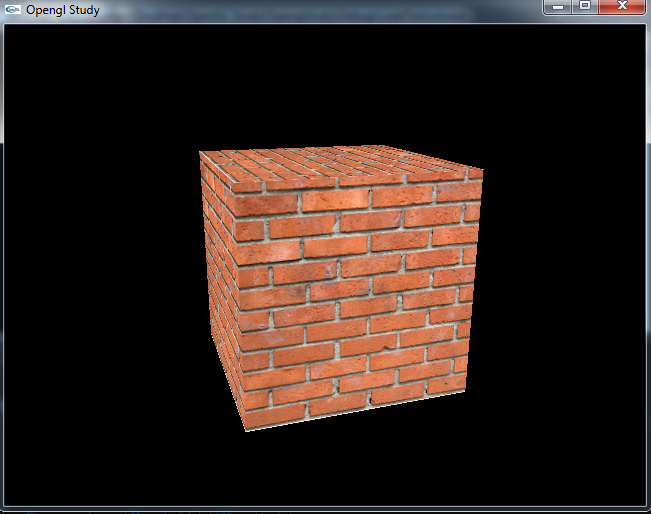
Ví dụ:

|  |
| --- |
| glNewList (listIndex, GL\_COMPLE) ;  glPushMatrix () ;  glPushAttrib ( GL\_ CURRENT\_BIT) ;  glColor3f( 1.0, 0.0, 0.0) ;  glBegin ( BOLYGON );  glvertex2f ( 0.0, 0.0);  glvertex2f ( 1.0, 0.0);  glvertex2f ( 1.0, 1.0);  glEnd () ;  glTranslatef (1.5, 0.0, 0.0 );  glPopAttrib () ;  glPopMatrix () ;  glEndList (); |

### Dán Texture

#### Texture là gì?

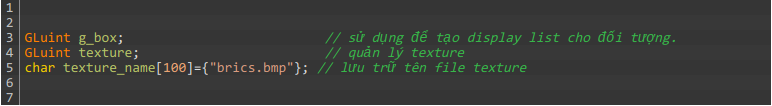
Hiểu đơn giản là chúng phủ vật liệu lên một đối tượng đồ họa 3d mà chúng ta tạo ra.Vật liệu ở đây chính là một tấm ảnh dạng bitmap và chúng được gọi là texture.  
Ví dụ tôi tạo ra một hình lập phương. Và tôi muốn phủ vật liệu cho nó là dạng gạch.Thì kết quả tạo ra sẽ như sau.

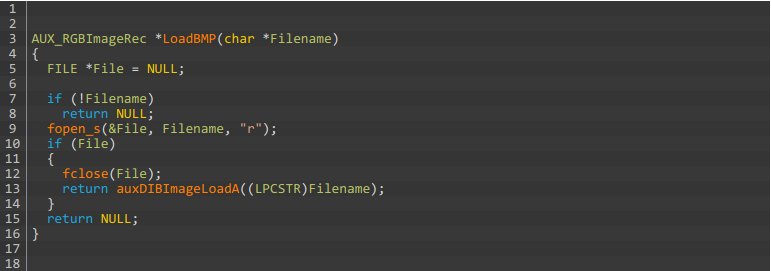
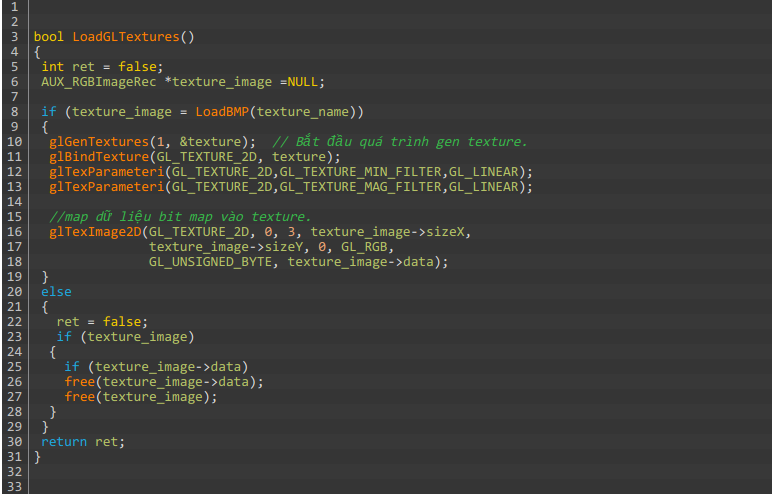
Như vậy tính thực tế được thể hiện rất rõ ở đây.Và texture giúp bạn mô phỏng thế giới thực gần như chân thực nhất.

Ví dụ khi bạn render một bức tường. Bạn có thể dùng texture để phủ vật liệu cho bức tường để nó giống thực tế hơn so khi mô phỏng.

#### Cách phủ texture

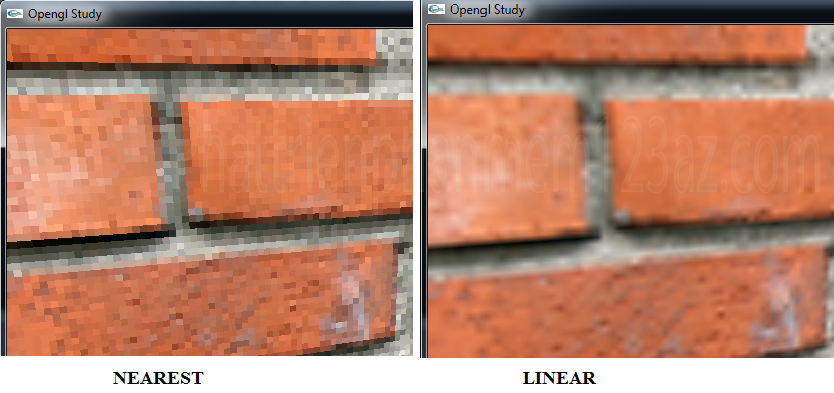
* Cài đặt thư viện glaux.
* Copy file bitmap vào cùng folder với source projection.
* Khai báo 3 biến global như sau:



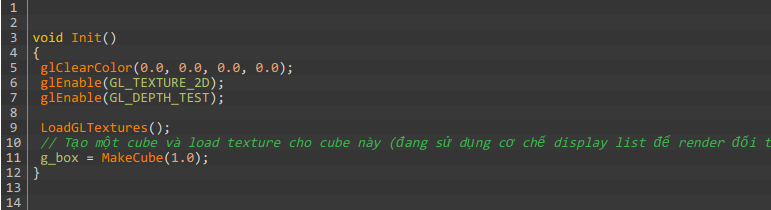
* Viết một hàm load bitmap sử dụng thư viện glaux.lib như sau:
* Viết một hàm load texture:
* Chế độ filter min mag trong câu lệnh glTexparameteri ở trên có hai loại như sau:

Sự khác nhau giữa hai kiểu này như sau:

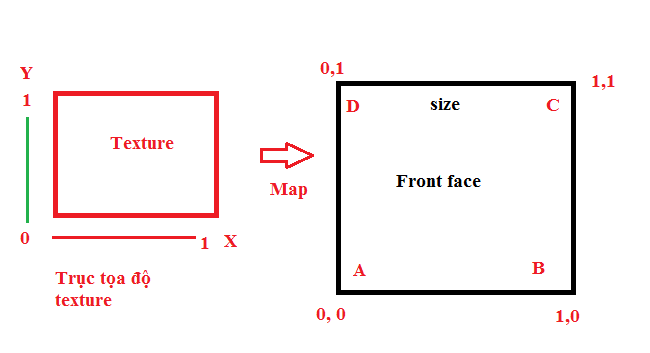
Khi view càng gần, thì NEAREST thể hiện là sự kết hợp bởi các hình vuông nhỏ, nó giống như điểm ảnh sắp xếp với nhau trên một bức ảnh.  
Còn LINEAR thì như một sự hòa trộn lại với nhau.  
Hãy xem hình vẽ dưới đây.



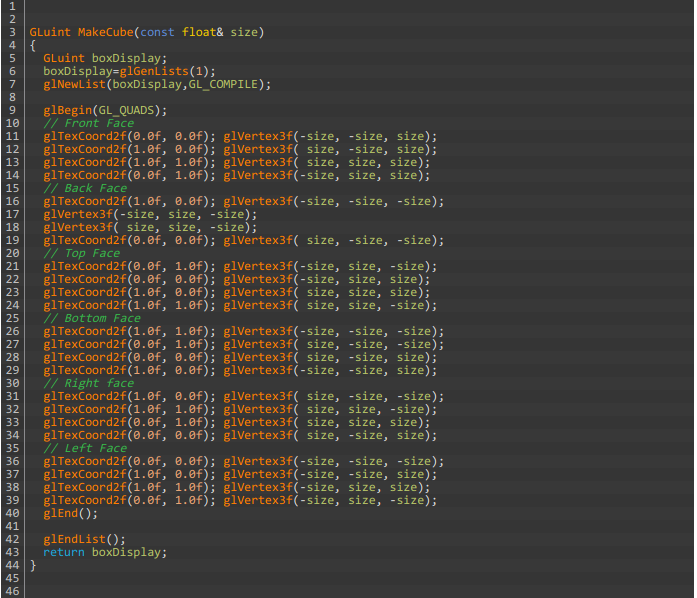
* Chúng ta enable chế độ load texture trong hàm Init của project OpenGL. Sau đó gọi hàm LoadTexture đã xây dựng ở trên



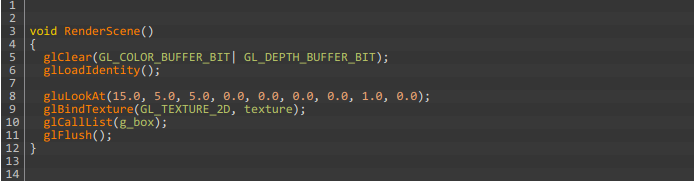
* Trong hàm MakeCube ở. Sử dụng hàm glTexcoord2f để map texture vào từng face của cube. Với cơ chế như sau:



Chúng ta sẽ thấy một texture vuông thì sẽ có tọa độ như ở trên. Góc ở mép dưới sẽ là gốc O. trục Y hương lên và trục X hướng ngang. Và toàn bộ size của texture được coi là mức đơn vị. Từ đó nó sẽ map tương ứng sang các face của đối tượng.

Và đây là hàm MakeCube:

* Trong hàm RenderScene:

Như vậy là chúng ta đã mapping texture vào một đối tượng 3D cơ bản.

### Framebuffer

* Là một phần của VRAM. Chứa bitmap data dùng để hiển thị lên màn hình.
* Framebuffer có thể được access thông qua các method sau:
* Được map địa chỉ vào address space của CPU và CPU write trực tiếp vào đó.
* CPU send command cho GPU (vd: thông qua OpenGL APIs), GPU write vào Framebuffer.
* Display (screen) sẽ phải quét Framebuffer để hiển thị (tần số quét 60Hz, 70Hz…).

### Quadrics

***Trong GLU có các hàm hổ trợ cho việc biểu diễn một số dạng 3D cơ bản như các khối cầu, cylinder, các đĩa… được gọi chung là Quadric***.

**Để sử dụng đối tượng Quadric, thực hiện theo các bước sau:**

* Tạo một đối tượng quadric:

GLUquadricObj\* gluNewQuadric(void): tạo một đối tượng quadric và trả về một pointer.

* Chỉ định những thuộc tính cho đối tượng quadric bằng:

*void* ***gluQuadricOrientation*** *(GLUquadricObj \**qobj*, GLenum* orientation*)*,

*void* ***gluQuadricDrawStyle*** *(GLUquadricObj \**qobj*, GLenum* drawStyle*)*,

*void* ***gluQuadricNormals*** *(GLUquadricObj \**qobj*, GLenum* normals*)*

* Quản lý sự xuất hiện lỗi trong quá trình biểu diễn.

*void* ***gluQuadricCallback*** *(GLUquadricObj \**qobj*, GLenum* which*, void (\**fn*)()).*

* Gọi những routines biểu diễn những đối tượng mong muốn.

*void* ***GluSphere****(GLUquadricObj \*qobj, Glduoble radius, Glint slices, Glint stacks ); //* Vẽ hình cầu bán kính radius tâm tại gốc.

*void* ***gluCylinder****( GLUquadricObj \*qobj, Gldouble basicRadius, Gldouble topRadius, Gldouble height, Glint slices, Glint stacks);* // Vẽ hình trụ có trục là trục z một mặt tiếp xúc với mặt z=0 và có chiều cao là height

*void* ***gluDisk****(GLUquadricObj \*qobj, Gldouble innerRadius, Gldouble outerRadius, Glint slices, Glint stacks);* // Vẽ hình đĩa có bán kính đường tròn trong interRadius và bán kính đường tròn ngoài outerRadius.

*void* ***gluPartialDisk****(GLUquadric \*qobj, Gldouble innerRadius, Gldouble outerRadius, Glint slices, Glint rings, Gldouble starAngle, Gldouble sweepAngle);*

* Loại bỏ vùng void gluDeleteQuadric(GLUquadric\*quad);

### Chọn đối tượng

#### Giới thiệu

Việc cho phép người dùng chọn đối tượng bằng cách click chuột trên cửa sổ là một yêu cầu thiết yếu đối với các ứng dụng tương tác. Để thực hiện được những chức năng như vậy, trong OpenGL có sẵn một chế độ là Selection.

Có 2 công đoạn lớn chúng ta cần phải làm:

(1) Thực hiện các thao tác vẽ trong chế độ render (đây là điều mà các phần trước đã bàn tới).

(2) Thực hiện các thao tác vẽ trong chế độ selection (giống hoàn toàn như trong công đoạn (1), kết hợp với một số thao tác đặc trưng trong chế độ selection.

Công đoạn (1) là các thao tác để biến đổi các đối tượng trong không gian về các pixel và sau đó hiển thị lên màn hình. Công đoạn 2, gần như ngược lại, chương trình xác định xem pixel mà người dùng tương tác (ví dụ như nhấn chuột trái) thuộc đối tượng nào.

Để chuyển đổi qua lại giữa các công đoạn (hay chế độ), chúng ta dùng hàm

GLint **glRenderMode**(GLenum mode);

trong đó mode là GL\_RENDER hoặc GL\_SELECT (mode còn có thể là GL\_FEEDBACK nhưng ở đây chúng ta sẽ không xét tới).

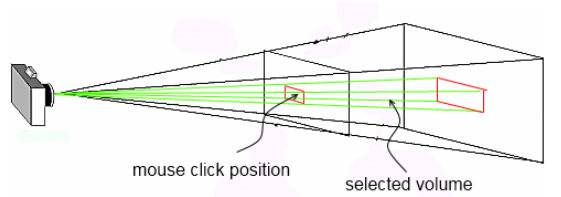
#### Các thao tác trên chế độ selection

Trước tiên chúng ta cần kích hoạt chế độ selection

glRenderMode(GL\_SELECT)

##### Xác định vùng chọn

Ví dụ về chọn đối tượng bằng click chuột được cho như hình dưới đây:



Việc xác định vùng chọn tương tự như là việc xác định khối nhìn, tức là chúng ta sẽ thao tác trên phép chiếu (projection).

Thao tác tổng quát như sau:

|  |
| --- |
| glMatrixMode (GL\_PROJECTION);  glPushMatrix ();  glLoadIdentity ();  gluPickMatrix (...);  gluPerspective, glOrtho, gluOrtho2D, or glFrustum  /\* ... \*/  glPopMatrix(); |

Trong đó, void **gluPickMatrix**(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]); là hàm xác định vùng quan tâm trong viewport (ví dụ như xung quanh vùng click chuột) với:

* x, y, width, height) là tham số xác định quan tâm trên viewport.
* viewport[4] là mảng 4 phần tử chứa 4 tham số của viewport, có thể dùng hàm glGetIntegerv(GL\_VIEWPORT, GLint \*viewport) để lấy ra.

##### Thiết lập đối tượng và danh tính cho đối tượng

Để phân biệt được các đối tượng với nhau, OpenGL cần phải đặt tên cho các đối tượng cần quan tâm. Việc đặt tên này có 3 điều đáng lưu ý:

(1) Tên là một số nguyên.

(2) Các đối tượng có thể mang cùng tên: đây là các đối tượng được gom vào cùng một nhóm được quan tâm, ví dụ như nhóm các hình cầu, nhóm các hình khối hộp,…

(3) Tên có thể mang tính phân cấp, thể hiện ở đối tượng được cấu thành từ nhiều thành phần khác nhau. Ví dụ như khi click vào một cái bánh của cái một cái xe hơi, chúng ta cần biết là cái bánh số mấy của cái xe hơi số mấy.

OpenGL có một stack giúp thao tác trên tên các đối tượng, với các hàm:

* void **glInitNames**(void) khởi tạo stack (stack lúc này rỗng)
* void **glPushName**(GLuint name) đặt tên của đối tượng cần xét vào trong stack
* void **glPopName**(void) lấy tên nằm ở đỉnh stack ra khỏi stack
* void **glLoadName**(GLuint name) thay nội dung của đỉnh stack

Việc sử dụng stack này giúp cho mục đích thứ (3) – xây dựng tên mang tính phân cấp. Mỗi lần thực thi glPushName(name) hoặc glLoadName(name) thì chương trình sẽ hiểu là các đối tượng được vẽ ở các dòng lệnh sau sẽ có tên là name và chúng là thành phần bộ phận của đối tượng có tên đặt ở ngay dưới đỉnh stack.

**Ví dụ:**

|  |  |
| --- | --- |
| **Xét đoạn mã giả sau** | **Stack** |
| glInitNames();  glPushName(1);  /\* vẽ đối tượng thứ nhất \*/  ...  glPushName(2);  /\* vẽ đối tượng thứ 2 \*/  ...  glPushName(4);  /\* vẽ đối tượng thứ 3 \*/  ... | nghĩa là đối tượng (4) là thành phần của đối tượng (2), và đối tượng (2) là thành phần của đối tượng (1). |

##### Truy vấn đối tượng trong vùng chọn

Để có thể truy vấn xem đối tượng nào được chọn, OpenGL xử lý như sau:

* Trước tiên sẽ đánh dấu mọi đối tượng nào có vùng giao với vùng chọn.
* Sau đó, với mỗi đối tượng có vùng giao, tên của nó và giá trị z nhỏ nhất, z lớn nhất của vùng giao sẽ được lưu trong hit records.
* Mọi truy vấn về đối tượng được chọn sẽ được thực hiện trên hit records.

Như vậy, dựa trên hit records chúng ta biết được các thông tin sau

(1) Số recods = số lượng đối tượng cần quan tâm nằm trong vùng chọn.

(2) Với mỗi record, chúng ta biết được các thông tin sau:

* Tên của đối tượng (bao gồm tên của tất cả các đối tượng mà nó là thành phần)
* z\_min và z\_max của vùng giao giữa đối tượng với vùng chọn (2 con số này nằm trong [0,1] và cần phải nhân với 231-1 (0x7fffffff) ).

Để khởi tạo hit records, chúng ta cần phải gọi hàm:

void **glSelectBuffer**(GLsizei size, GLuint \*buffer)

trong đó buffer chính là mảng chứa hit records.

**Chú ý:** thủ tục này phải được gọi trước khi chuyển sang chế độ GL\_SELECT.

## Xây dựng ứng dụng

### Giới thiệu

Hiện nay, sự phát triển của công nghệ hầu như có mặt ở khắp các lĩnh vực trong cuộc sống con người. Và việc áp dụng công nghệ vào giáo dục có thể mang lại nhiều thú vị và giúp ghi nhớ dễ dàng hơn cho học sinh.

Với những kiến thức đã được học trong môn Đồ Họa Máy Tính, cùng với việc tìm hiểu cách sử dụng OpenGL cho đối tượng 3D, nhóm đã quyết định tiến hành xây dựng ứng dụng “Mô phỏng Hệ Mặt Trời 3D” nhằm mục đích cho cung cấp các kiến thức cơ bản về vũ trụ (các hành tinh, quỹ đạo quay, lực hấp dẫn,…) giúp học sinh có cách tiếp cận tốt hơn về các kiến thức đã được học, đồng thời tăng cường quá trình tương tác giữa giáo viên và học sinh.

### Phát biểu bài toán

Mục đích của ứng dụng:

* Vận dụng các kiến thức đã tìm hiểu về OpenGL cho đối tượng 3D để xây dựng nên một ứng dụng cơ bản nhưng hữu ích.
* Tạo ra một mô hình 3D về Hệ Mặt Trời, với các kiến thức phổ thông đã được học để mô phỏng chính xác, chi tiết.

Lợi ích của ứng dụng

* Là một mô hình demo về Hệ Mặt Trời có thể dùng để giảng dạy trong trường học.
* Có thể được dùng làm một tài liệu tham khảo về mô phỏng vũ trụ.
* Cung cấp cho mọi người có cái nhìn trừu tượng về Hệ Mặt Trời.

Giới hạn bài toán:

* Ứng dụng có khả năng hiển thị tương đối về: đường kính, khoảng cách, quỹ đạo quay, và tốc độ quay của các hành tinh xung quanh Mặt Trời.
* Ứng dụng phải thể hiện đúng quỹ đạo quay của Trái Đất (tự quanh mình nghiên một góc 24,3o) và phải có Mặt Trăng quay xung quanh.
* Ứng dụng phải dán texture cho các hành tinh để có cái nhìn trực quan sinh động về Hệ Mặt Trời, đồng thời cung cấp một số thông tin thú vị về các hành tinh để thu hút người dùng.
* Ứng dụng có khả năng tương tác người dùng (thông qua bàn phím).
* Ứng dụng được thiết kế dựa vào API OpenGL, ngôn ngữ C++.

### Giải quyết bài toán

**Vật liệu tạo nên Hệ Mặt Trời:**

* Thành phần của Hệ Mặt Trời: hiển thị thông tin Mặt Trời, thứ tự của các hành tinh được sắp xếp tăng dần theo khoảng cách đến Mặt Trời và quỹ đạo quay của chúng (bao gồm quỹ đạo quay quanh Mặt Trời và quỹ đạo tự quay 360o quanh mình).
* Kích thước hành tinh: Hiển thị kích thước tương đối giữa các hành tinh so với kích thước thực tế.
* Khoảng cách từ hành tinh đến Mặt Trời: Hiển thị khoảng cách tương đối giữa các hành tinh so với tốc độ thực tế.
* Tốc độ quay của hành tinh: tốc độ tương đối so với thực tế.
* Quỹ đạo quay của hành tinh: Hiển thị mọi quỹ đạo trong Hệ Mặt Trời.
* Chế độ chiếu sáng: Mặt Trời sẽ là nguồn sáng duy nhất trong hệ.

**Kiến thức về OpenGL được sử dụng:**

* Vẽ đối tượng hình học 3D (Sphere, Circle).
* Các phép biến đổi Affine(Translate, Rotate).
* Phép chiếu Perpective.
* Dán Texture cho đối tượng 3D.
* Xử lý sự kiện bàn phím.

**Công đoạn để tạo Hệ Mặt Trời:**

1. Khai báo các thuộc tính cần thiết cho các hành tinh về: khoảng cách tới Mặt Trời (tính bằng km), bán kính (tính bằng km), và tốc độ quay (tính bằng ngày ở Trái Đất).
2. Khai báo các giá trị scale để làm giảm các kích thước trong Hệ Mặt Trời nhưng vẫn đảm bảo tính tương đối của các hành tinh trong Hệ Mặt Trời.
3. Tạo Mặt Trời là một quả cầu rắn với kích thước tương đối(nhưng đảm bảo to nhất trong Hệ Mặt Trời), và được đặt ở trung tâm góc tọa độ (0,0,0): để cho Mặt Trời trở thành trung tâm mà các hành tinh khác sẽ quay quanh.
4. Tạo các hành tinh khác: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune là một quả cầu rắn đặc với kích thước tương đối, và được đặt trong hệ tọa độ theo tỉ lệ khoảng cách thực tế của chúng.
5. Xác định tốc độ quay cho mỗi hành tinh, bao gồm: tốc độ mà nó quay quanh bản thân và quay quanh Mặt Trời.
6. Vẽ thêm Moon ứng Earth và xem Earth là trung tâm để quay quanh. Lưu ý về quỹ đạo tự quay quanh mình của Trái Đất.
7. Tạo quỹ đạo quay của các hành tinh và Moon, sao cho đảm bảo rằng chúng vẫn luôn quay quanh quỹ đạo của mình.
8. Tạo thêm chức năng bàn phím để điều chỉnh chuyển động của camera giúp tương tác người dùng.
9. Load texture cho các hành tinh để tạo sự chân thật cho ứng dụng.
10. Điều chỉnh ánh sáng để tạo hiệu ứng 3D.
11. Cập nhật trạng thái của toàn bộ Hệ Mặt Trời sau những khoảng thời gian khác nhau.
12. Xử lý các sự kiện bàn phím từ người dùng.

|  |  |
| --- | --- |
| **Tương tác người dùng** | **Sơ đồ mô phỏng chương trình ứng dụng “Hệ Mặt Trời 3D”** |
| **Camera:**  **a**: dịch sang trái  **d**: dịch sang phải  **w**: dịch lên  **s**: dịch xuống  **e**: đi lên  **q**: đi xuống  **Điểm nhìn:**  **j**: dịch sang trái  **l**: dịch sang phải  **i**: dịch lên  **k**: dịch xuống  **u**: đi lên  **o**: đi xuống  **Hiển thị thông tin:**  **f**: Hiển thị thông tin cơ bản của toàn bộ Hệ mặt Trời  **Các số từ 0 đến 9**: hiển thị thông tin của các hành tinh tương ứng với khoảng cách tăng dần tính từ Mặt Trời.  **n**: Quay lại mô hình Hệ Mặt Trời.  **h**: hiện hướng dẫn sử dụng.  **p**: hiện/ẩn quỹ đạo quay  **c**: hiện/ẩn trục tọa độ  **Thay đổi tỉ lệ thước hành tinh:**  **[** : tăng kích thước  **]**: giảm kích thước  **Thay đổi tỉ lệ tốc độ quay:**  = : tăng tốc độ  **-**: giảm tốc độ | Bắt đầu  Khởi tạo các giá trị ban đầu của OpenGL để vẽ HMT  Xử lý các sự kiện bàn phím và cập nhật thay đổi nếu có.  Người dùng nhấn phím  Vẽ Hệ Mặt Trời ra màn hình  Kết thúc  Thiết lập cửa sổ Window |

### Thực nghiệm

***Lưu ý quan trọng***: Source code sử dụng thư viện glut, nếu mở source code lên bị lỗi thì cách fix là add 2 file (glut32.h và glut32.lib) cho source code.

#### Giao diện

**Khởi động ứng dụng:**



Figure 1. Solar System

**Xem thông tin Solar System:**

Bấm h: hiển thị bảng hướng dẫn sử dụng/ngược lại.

Bấm p: hiển thị quỹ đạo các hành tinh trong Solar System/ngược lại.

Bám c: hiển thị trục tọa độ của ứng dụng/ngược lại.

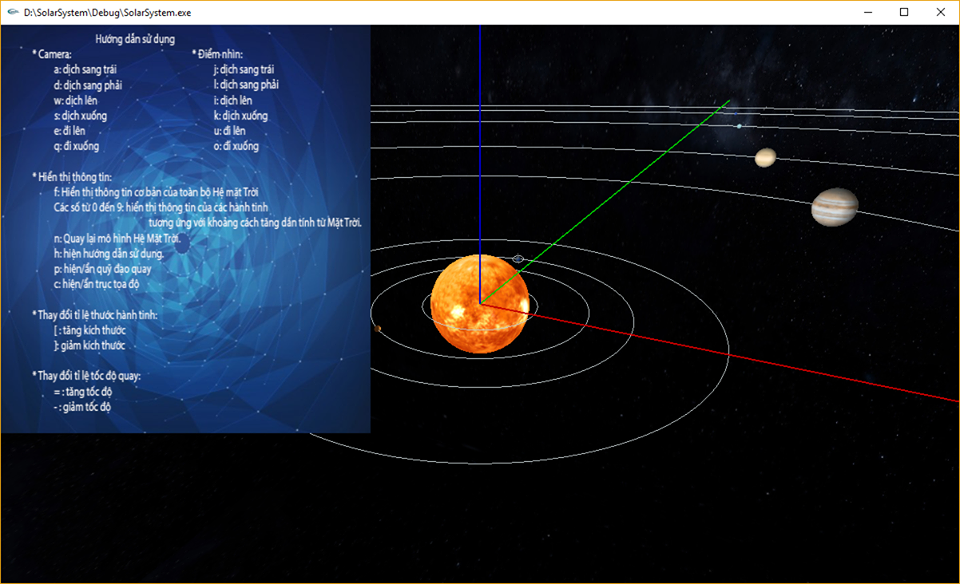


Figure 2. Thông tin Solar System

**Xem thông tin Planet:**

Bấm các số tương ứng từ 1 đến 9: để xem thông tin các hành tinh.

Bấm n: để quay lại Solar System.



Figure 3. Thông tin Trái Đất

#### Cài đặt chi tiết

**Kích thước của hành tinh: [3]**

|  |  |  |
| --- | --- | --- |
| Planets | Bán kính | |
| Thực tế  (km) | OpenGL |
| Sun | 695500 | 0.5 |
| Merury | 2440 | 0.0122 |
| Venus | 6052 | 0.03026 |
| Earth | 6371 | 0.031855 |
| Mars | 3389 | 0.0169745 |
| Jupiter | 69911 | 0.349555 |
| Saturn | 58232 | 0.29116 |
| Uranus | 25362 | 0.12681 |
| Neptune | 24622 | 0.12311 |
| Pluto | 1137 | 0.005685 |

Bán kính thực tế của các hành tinh đến Mặt Trời nhóm đã lấy dữ liệu từ trang web chính thức của [nasa](https://pds.jpl.nasa.gov/planets/). Với kích thước tương ứng đó, nhóm đã tạo ra một bảng bán kính mới để dùng trong OpenGL cho ứng dụng của mình, bằng cách lấy bán kính thực tế chia cho 2000000 để làm giảm bán kính mô phỏng, vì vậy tỉ lệ là 2000000:1. Nhìn vào dữ liệu của bảng trên ta thấy rằng bán kính trong OpenGL của các hành tinh (không phải Mặt Trời) chỉ đạt lớn nhất là 0.349555(Jupiter), nên nhóm có thể mô phỏng Mặt Trời với bán kính 0.5 vẫn đảm bảo rằng Mặt Trời là tô nhất trong hệ.

**Khoảng cách từ hành tinh đến Mặt Trời:[3]**

|  |  |  |
| --- | --- | --- |
| Planets | Khoảng cách đến Sun | |
| Thực tế  (km) | OpenGL |
| Sun | 0 | 0 |
| Merury | 57910000 | 0.591 |
| Venus | 108200000 | 1.820 |
| Earth | 149600000 | 1.496 |
| Mars | 227939100 | 2.27939 |
| Jupiter | 778500000 | 7.785 |
| Saturn | 1433000000 | 14.33 |
| Uranus | 2877000000 | 28.77 |
| Neptune | 4503000000 | 45.03 |
| Pluto | 5906380000 | 59.0638 |

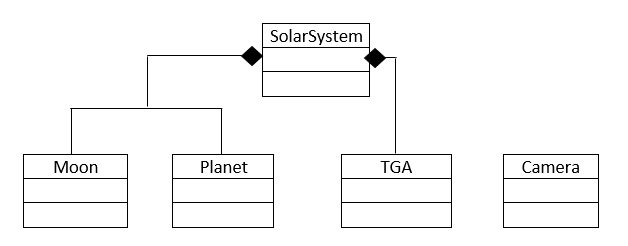
Khoảng cách thực tế từ các hành tinh đến Mặt Trời nhóm đã lấy dữ liệu từ trang web chính thức của [nasa](https://pds.jpl.nasa.gov/planets/). Với kích thước tương ứng đó, nhóm đã tạo ra một bảng kích thước mới để dùng trong OpenGL cho ứng dụng của mình, bằng cách lấy khoảng cách thực tế chia cho 108 để làm giảm kích thước mô phỏng. Vì vậy tỉ lệ là 108:1.

**Tốc độ quay của hành tinh:[3]**

|  |  |  |
| --- | --- | --- |
| Planets | Tốc độ quay | |
| Tốc độ quay quanh Mặt Trời  (ngày Trái Đất) | Tốc độ tự quay 360o  (Ngày Trái Đất) |
| Sun | 1 | 695500 |
| Merury | 88 | 58.6 |
| Venus | 224.65 | 243 |
| Earth | 365 | 1 |
| Mars | 686 | 1.03 |
| Jupiter | 4332 | 0.4139 |
| Saturn | 10759 | 0.44375 |
| Uranus | 30685 | 0.718056 |
| Neptune | 60188 | 0.6713 |
| Pluto | 90616 | 6.39 |

Tốc độ quay thực tế từ các hành tinh nhóm đã lấy dữ liệu từ trang web chính thức của [nasa](https://pds.jpl.nasa.gov/planets/). Mỗi hành tinh sẽ đi hết quỹ đạo của mình (đường tròn tâm O, bán kính là khoảng cách từ hành tinh đó đến Mặt Trời) với tốc độ quay tương ứng của mình, đồng thời tự quay 360o quanh bản thân.

**Sơ đồ lớp ứng dụng:[5]**



* Lớp SolarSystem là lớp chủ đạo của ứng dụng, có tác dụng lưu trữ thông tin toàn bộ Hệ Mặt Trời và cập nhật trạng thái theo thời gian.
* Các lớp Moon và Planet là các lớp riêng lẽ dùng để lưu trữ thông tin của một mặt trăng hay một hành tinh cụ thể.
* Lớp TGA là lớp dùng để đọc ảnh có đuôi “.tga” để dán texture cho các đối tượng trong hệ mặt trời.
* Lớp Camera dùng để lưu thông tin camera, và thiết lập các thông số để điều khiển camera hay điểm nhìn từ người dùng.

|  |
| --- |
| **Lớp Moon** |
| /\*Mỗi một hành hành tinh có thể có các moon quay quanh nó.  Việc này tương tự như các hành tinh có thể quay quanh Mặt Trời.  Mặt trăng không phải là 1 hành tinh trong vũ trụ, nó không quay quanh Mặt Trời, mà lại quay quanh Trái Đất\*/  class Moon {  private:  /\* Đơn vị tính: km  - distanceFromPlanet: Khoảng cách từ moon đến Planet  - radius: Bán kính của Moon  \*/  float distanceFromPlanet, radius;  /\* Đơn vi tính: ngày ở Trái Đất  - orbitTime: Thời gian để Moon hoàn thành 1 chu kỳ quỹ đạo quay quanh Planet  - rotationTime: Thời gian để Moon tự quay 360 độ quanh nó  \*/  float orbitTime, rotationTime;  // Ảnh được phủ lên bề mặt Moon  GLuint textureHandle;  // Tọa độ (x,y,z) của Moon so với Planet mà nó quay quanh  float position[3];  // Góc mà Moon tự quanh quanh nó  float rotation;  public:  // Contructer  Moon(float distanceFromPlanet, float orbitTime, float rotationTime, float radius, GLuint textureHandle);  // Cập nhật lại trạng thái của Moon sau khoảng thời gian time tính từ thời điểm ban đầu (= 0)  void updateStatus(float time);  // Vẽ Moon ra không gian vũ trụ  void render();  // Vẽ quỹ đạo quay của Moon ra không gian vũ trụ  void renderOrbit();  }; |

|  |
| --- |
| **Lớp Planet** |
| class Planet {  private:  /\*Đơn vị tính: km  distanceFromSun: Khoảng cách từ Planet đến Sun  radius: Bán kính của Planet  \*/  float distanceFromSun, radius;  /\* Đơn vị tính: ngày ở Trái Đất  - orbitTime: Thời gian để Planet hoàn thành 1 chu kỳ quỹ đạo quay quanh Mặt Trời  - rotationTime: Thời gian để Planet tự quay 360 độ quanh nó  \*/  float orbitTime, rotationTime;  // Ảnh được phủ lên bề mặt Planet  GLuint textureHandle;  // Tọa độ (x,y,z) của Planet so với Sun mà nó quay quanh  float position[3];  // Góc mà Planet tự quanh 360 độ quanh nó  float rotation;  // Danh sách các Moon xung quanh Planet  std::vector<Moon> moons;  public:  // Constructor  Planet(float distanceFromSun, float orbitTime, float rotationTime, float radius, GLuint textureHandle);  // Cập nhật lại trạng thái của Planet sau khoảng thời gian time tính từ thời điểm ban đầu (= 0)  void updateStatus(float time);  // Vẽ Planet ra không gian vũ trụ  void render();  // Vẽ quỹ đạo quay quanh Sun của Planet ra không gian vũ trụ  void renderOrbit();  // Các getting  void getPosition(float vec[3]);  float getRadius();  // Thêm 1 Moon vào danh sách moons quay quanh Planet  void addMoon(float distanceFromPlanet, float orbitTime, float rotationTime, float radius, GLuint textureHandle);  // Vẽ hành tinh tự quay 360 độ quanh mình mà không quay quanh Mặt Trời  void renderOnly();  }; |

|  |
| --- |
| **Lớp TGA** |
| // This is a class that loads TGA files into opengl textures  class TGA  {  private:  // the handle for the texture in opengl  GLuint textureHandle;  public:  // Constructs and loads a TGA into opengl from the given image file path  TGA(char\* imagePath);  // Returns the handle to the texture created from the image, for binding to opengl  GLuint getTextureHandle(void);  }; |

|  |
| --- |
| **Lớp Camera** |
| class Camera {  private:  // Vị trí của Camera trong không gian  float position[3];  // Điểm nhìn của Camera  float pointView[3];  // Tốc độ di chuyển của Camera hay điểm nhìn  float speedMove;  // Bán kính, tốc độ, góc khi di chuyển camera xung quanh điểm nhìn  float radius;  float speedRotate;  float angle;  public:  Camera(); // Contructer  void set(); // Đặt Camera vào không gian  void update(); // Cập nhật góc quay, bánh kính quay  void speedUp(); // Tăng tốc độ  void speedDown(); // Giảm tốc độ  void forwardCamera(); // Camera tiến về trước  void backwardCamera(); // Camera lùi về sau  void rightCamera(); // Camera dịch sang phải  void leftCamera(); // Camera dịch sang trái  void upCamera(); // Camera dịch lên trên  void downCamera(); // Camera dịch xuống dưới  void forwardPointView(); // Dịch điểm nhìn tiến về trước  void backwardPointView(); // Dịch điểm nhìn lùi về sau  void rightPointView(); // Dịch điểm nhìn sang phải  void leftPointView(); // Dịch điểm nhìn sang trái  void upPointView(); // Dịch điểm nhìn lên trên  void downPointView(); // Dịch điểm nhìn xuống dưới  }; |

|  |
| --- |
| **Lớp SolarSystem** |
| class SolarSystem {  private:  // Lưu lại 9 Planet trong không gian vũ trụ  std::vector<Planet> planets;  public:  // Constructer  SolarSystem();  // Cập nhật lại trạng thái của Solar System sau khoảng thời gian time tính từ thời điểm ban đầu (= 0)  void updateStatus(float time);  // Thêm 1 Planet vào Solar System  void addPlanet(float distanceFromSun, float orbitTime, float rotationTime, float radius, GLuint textureHandle);  // Thêm 1 Moon của 1 Planet trong Solar System  void addMoon(int planetIndex, float distanceFromPlanet, float orbitTime, float rotationTime, float radius, GLuint textureHandle);  // Vẽ Solar System ra không gian 3D  void render();  // Vẽ quỹ đạo của tất cả Planets trong Solar System  void renderOrbits();  // Lấy tọa độ của 1 Planet thứ index lưu vào vec[3]  void getPlanetPosition(int index, float vec[3]);  // Lấy bán kính của 1 Planet thứ index  float getRadiusOfPlanet(int index);  // Vẽ duy nhất hành tinh vị trí index ra màn hình  void renderPlanet(int index);  }; |

### Kết luận

* Nhóm đã làm đúng các yêu cầu mà tự mình đặt ra bằng việc vận dụng các kiến thức đã tìm hiểu về OpenGL.
* Thực tế, nhóm cũng đã thấy một số hạn chế về chức năng, độ chính xác về quỹ đạo quay của các hành tinh xung quanh Mặt Trời, cũng như khả năng tương tác của ứng dụng với người dùng còn khá hạn chế (thiếu thao tác chuột, mô phỏng cấu trúc của hành tin,…) so với các ứng dụng cùng chủ đề khác.
* Nhóm mong muốn nếu có thời gian sẽ phát triển tốt ứng dụng của mình hơn để cố gắng khắc phục những điểm yếu ở trên để tạo nên một ứng dụng hữu ích, mô phỏng chính xác nhất có thể, giao diện đẹp, dễ sử dụng và hoàn toàn miễn phí cho giáo dục.

# Experimental Results

* Qua việc tìm hiểu về cách sử dụng OpenGL cho 3D, nhóm đã nhận thấy được sự tiện ích, cũng như việc sử dụng OpenGL là không quá khó cho một người đã có kiến thức về lập trình và Đồ Họa Máy Tính. Chúng ta hoàn toàn có thể sáng tạo ra những ứng dụng đồ họa 3D hữu ích phục vụ trong mọi lĩnh vực trong cuộc sống như: mô tả cơ thể người trong y học, phim hoạt hình 3D, mô phỏng thế giới tự nhiên, ứng dụng phục vụ cho các ngành kỹ thuật, mỹ thuật,…
* Ứng dụng mô phỏng Hệ Mặt Trời của nhóm tuy không phải là một ý tưởng sáng tạo, cũng không phải là một ứng dụng vượt trội so với các ứng dụng khác mà cùng chủ đề, nhưng ứng dụng này đã phần nào thể hiện được sự hữu ích của Đồ Họa Máy Tính trong thời buổi hiện nay, cũng như sự tiện dụng về lập trình của OpenGL trong việc xây dựng một ứng dụng đồ họa 3D trên môi trường Window.
* Mong muôn của nhóm, nếu có thêm thời gian sẽ xây dựng ứng dụng trở nên hoàn thiệt hơn về nhiều mặt: tương tác người dùng, mô phỏng chính xác nhất có thể, cung cấp nhiều thông tin thú vị về Hệ Mặt Trời.

# References

[1] Joey de Vries, Learn OpenGL, An offline transcript of learnopengl.com, 2015

[2] Mark Segal, Kurt Akeley, The OpenGLR Graphics System, The Khronos Group, 2010

[3] <https://pds.jpl.nasa.gov/planets/>

[4] <https://www.solarsystemscope.com/textures/>

[5] <https://github.com/RyanPridgeon/solarsystem>

[6] <https://thuthuattienich.vn/thu-thuat/opengl-la-gi>

[7] <http://glprogramming.com/red/>

[8] <https://phattrienphanmem123az.com/lap-trinh-opengl-cpp/opengl-cpp-bai-8-load-texture.html>

[9] <https://www.cppdeveloper.com/best-practices/mot-so-khai-nhiem-trong-lap-trinh-graphic/>