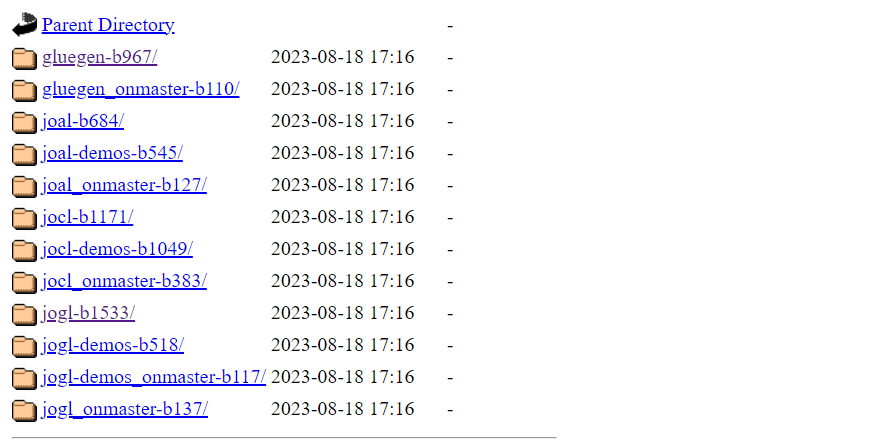
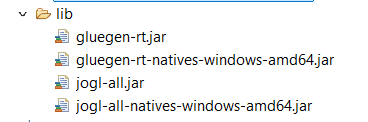
# Cài đặt thư viện JOGL

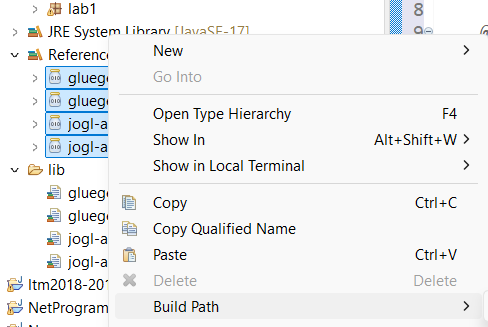
* Tải file cần thiết: link (<https://jogamp.org/deployment/autobuilds/master/?C=M;O=D>)
* Tải các file: gluegen và jogl



* Giải nén và copy các file sau vào /lib trong eclipse:



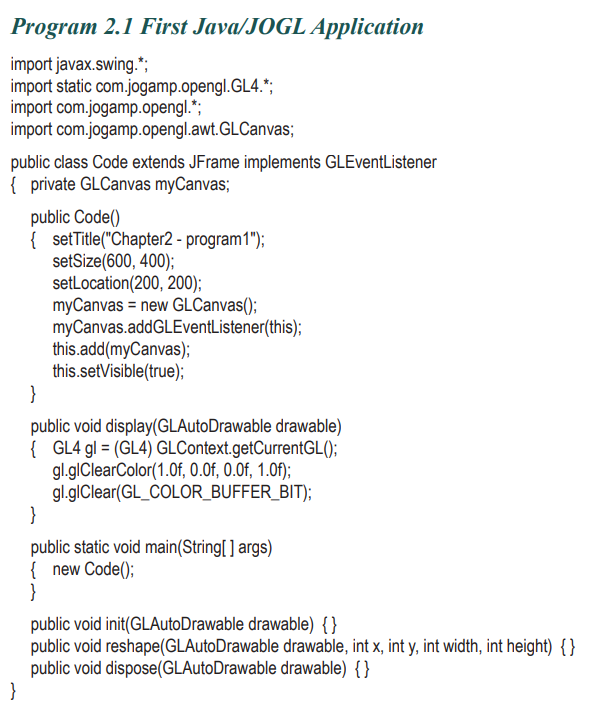
* Chọn tất cả và Build path:



# Chapter 2: THE OPENGL GRAPHICS PIPELINE

## 2.1. The OpenGL Pipeline

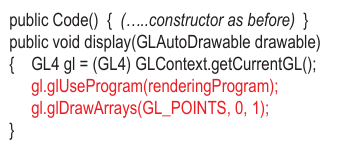
### 2.1.1. Java/JOGL Application

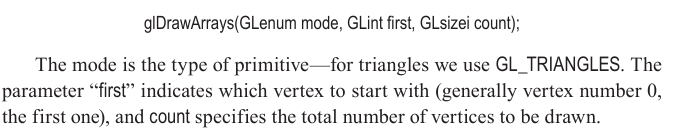


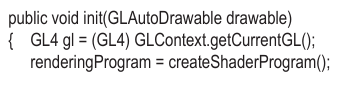
* Để dùng OpenGL cần chú trọng implement các phương thức như: display(), init(), reshape() và dispose(). Khi chương trình được chạy thì sẽ gọi callback các phương thức này.
* Phương thức display(): nơi chứa các code để vẽ lên GLCanvas
* Phương thức init(): giúp khởi tạo vùng có thể vẽ, trong ví dụ là GLCanvas. Trong ví dụ init() không để xử lý gì, nhưng hầu hết các chương trình là nơi sẽ đọc code GLSL, load mô hình 3D,…
* Phương thức reshape() được gọi khi GLCanvas được resize.
* Phương thức dispose() được gọi khi thoát ứng dụng.

### 2.1.2. Vexter and Fragment Shaders

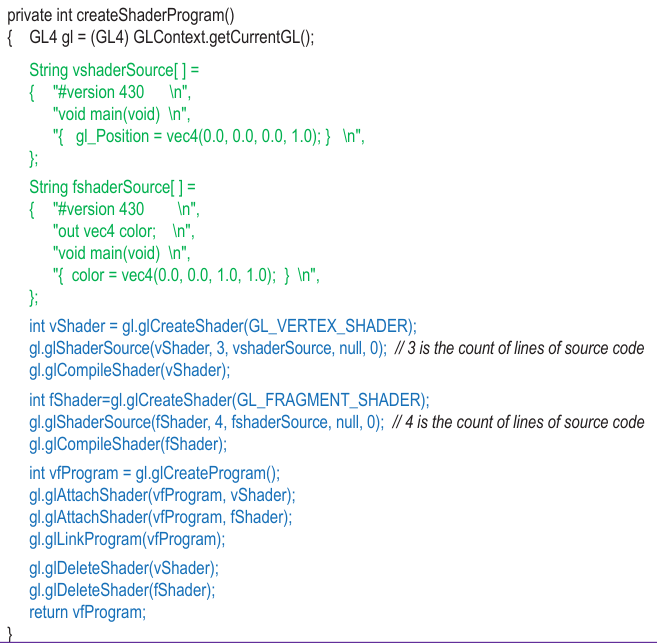
* **Vertex shader** làm việc với các điểm đỉnh và xử lý các biến đổi không gian.
* **Fragment shader** làm việc với các đoạn của pixel và quyết định cách chúng được tô màu, đổ bóng trên màn hình.











- gl.glCreateShader(): giúp tạo loại đối tượng shader (ban đầu là object rỗng) mà mình muốn (GL\_VERTEX\_SHADER hoặc GL\_ FRAGMENT\_SHADER). Hàm trả về ID kiểu int là tham chiếu đến shader đó.

- gl.glShaderSource(): load code GLSL từ mảng chuỗi string vào đối tượng shader rỗng vừa tạo phía trên.

glShaderSource() có 5 tham số:

1. đối tượng shader mà để lưu trữ shader muốn load lên.
2. Số lượng strings trong shader source code.
3. Mảng string chứa source code.

Và 2 tham số không dùng tới.

- glCompileShader(): tạo shader.

- glCreateProgram(): tạo chương trình opengl sẽ chứa các shader vừa compile phía trên, hàm trả về id của program.

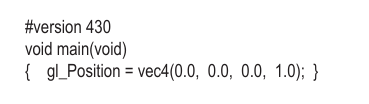
- glAttachShader(): sẽ gán các shader vào program vừa tạo.

- glLinkProgram(): yêu cầu glsl đảm bảo tương thích.

- Sau khi init() hoàn thành thì display() được gọi 1 cách tự động.

- glUseProgram(): sẽ load program chứa 2 shader đã tạo vào OpenGL pipeline stages (vào GPU), glUseProgram() không chạy shader mà chỉ load vào trong gpu.

- Tìm hiểu và vertex shader:



- dòng đầu tiền chỉ ra version của OpenGL.

- dòng tiếp theo là hàm main.

- Mục đích của vertex shader là gửi 1 vecto xuống pipeline.

- biến gl\_Position được sử dụng để set vị trí tọa độ điểm của vecto trong không gian 3D và nó được gửi vào stage kế tiếp của pipeline.

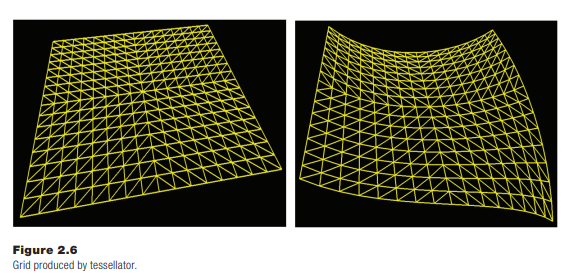
- vec4 sử dụng giữ 1 4-tuple, phù hợp cho tọa độ điểm.

- Mục đích của fragmen shader là thiết lập màu RGB cho pixel sẽ hiển thị, vec4 3 giá trị đầu là giá trị rgb và cuối là opacity.

- keyword “out” chỉ là biến sẽ là 1 output.

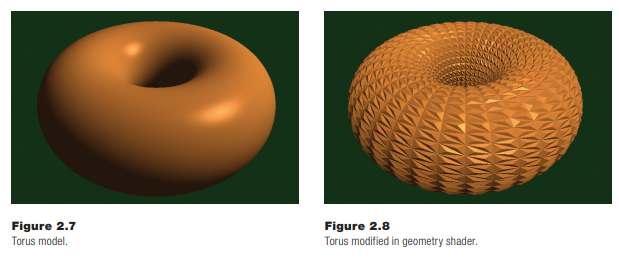
### 2.1.3. Tessellation

Trong đồ họa máy tính là quá trình chia nhỏ bề mặt của mô hình 3D thành các mảnh nhỏ hơn, thường là các tam giác, để tăng độ chi tiết và cải thiện chất lượng hiển thị. Nó giúp tạo ra các hiệu ứng bề mặt phức tạp, như chi tiết gồ ghề hoặc các bề mặt cong mịn hơn, bằng cách chia nhỏ hình học tùy theo mức độ cần thiết.



### 2.1.4. Geometry Shader

Là một giai đoạn trong pipeline đồ họa của GPU, nằm giữa Vertex Shader và Fragment Shader. Nó cho phép xử lý hình học, chẳng hạn như điểm, đường thẳng hoặc tam giác, và có thể tạo ra, thay đổi hoặc loại bỏ các hình học đó trước khi chúng được raster hóa (biến đổi thành các pixel để hiển thị trên màn hình).



### 2.1.5. Rasterization

Là quá trình chuyển đổi các đối tượng hình học (như điểm, đường thẳng, tam giác) trong không gian 3D thành các điểm ảnh (pixel) trên màn hình 2D để hiển thị hình ảnh. Đây là một bước quan trọng trong pipeline đồ họa của GPU, giúp biến đổi các mô hình 3D thành hình ảnh 2D mà chúng ta có thể nhìn thấy trên màn hình.

### 2.1.6 Fragment Shader

Là một giai đoạn trong pipeline đồ họa của GPU, chịu trách nhiệm xử lý các "fragment" (phân đoạn) để xác định màu sắc cuối cùng của từng pixel trên màn hình. Fragment Shader thực hiện các tính toán để xác định giá trị màu, độ trong suốt, và các thuộc tính khác của mỗi fragment dựa trên các thông tin từ giai đoạn rasterization, như tọa độ, màu sắc, và ánh sáng.

### 2.1.7 Pixel Operations

Là các thao tác được thực hiện trên các pixel hoặc fragment (phân đoạn) trong quá trình hiển thị đồ họa. Đây là các bước xử lý sau khi các pixel đã được tính toán bởi Fragment Shader nhưng trước khi ghi vào bộ đệm khung (framebuffer) để hiển thị lên màn hình. Các thao tác này bao gồm một loạt các kiểm tra và thay đổi để xác định xem và làm thế nào các pixel sẽ được lưu trữ trong bộ đệm khung.

## 2.2. DETECTING OPENGL AND GLSL ERRORS

## 2.3. READING GLSL SOURCE CODE FROM FILES

## 2.4. BUILDING OBJECTS FROM VERTICES

## 2.5. ANIMATING A SCENE

# Chapter 3: Mathematical Foundations

## 3.1. 3D COORDINATE SYSTEMS

## 3.2. POINTS

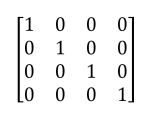
- Điểm 3D thường xác định dưới dạng x, y, z sử dụng trong tọa độ (x, y, z)

- Tuy nhiên để dễ dàng tính toán thì sử dụng tọa độ Homogeneous (Tọa độ đồng nhất, thêm 1 chiều có giá trị bằng 1). VD: (2, 8, 3) thì sẽ thành (2, 8, 3, 1).

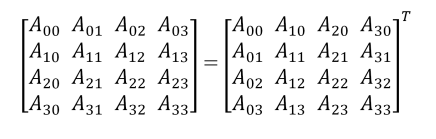
- Trong GLSL sử dụng vec4 để biểu diễn điểm. JOML là Vector3f và Vetor4f.

## 3.3. MATRICES

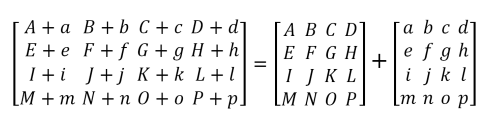
- Ma trận đơn vị: giá trị trên đường chéo 1 các vị trí khác là 0.



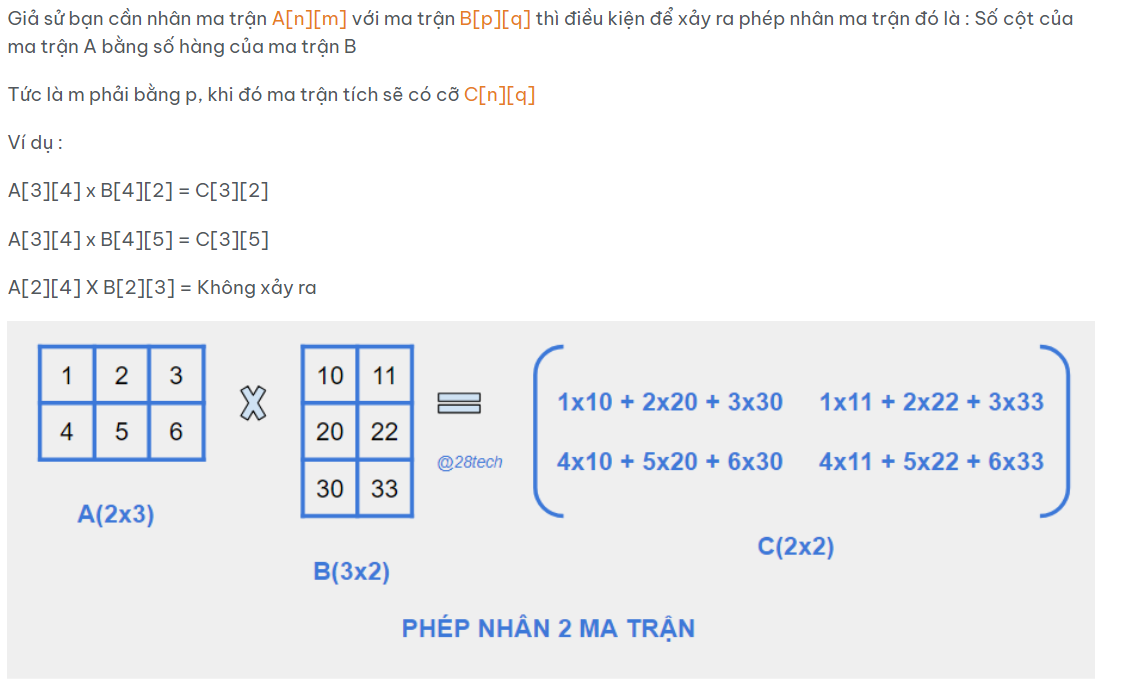
* Ma trận chuyển vị: Cầm góc trái dưới lật lên như lật trang sách.



* Phép cộng ma trận:



* Nhân ma trận:



* Ma trận nghịch đảo(khả nghịch):

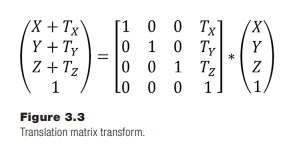


## 3.4. TRANSFORMATION MATRICES

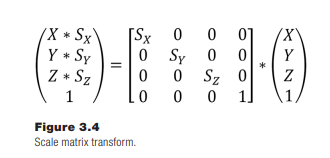
Các phép biến đổi thường dung:

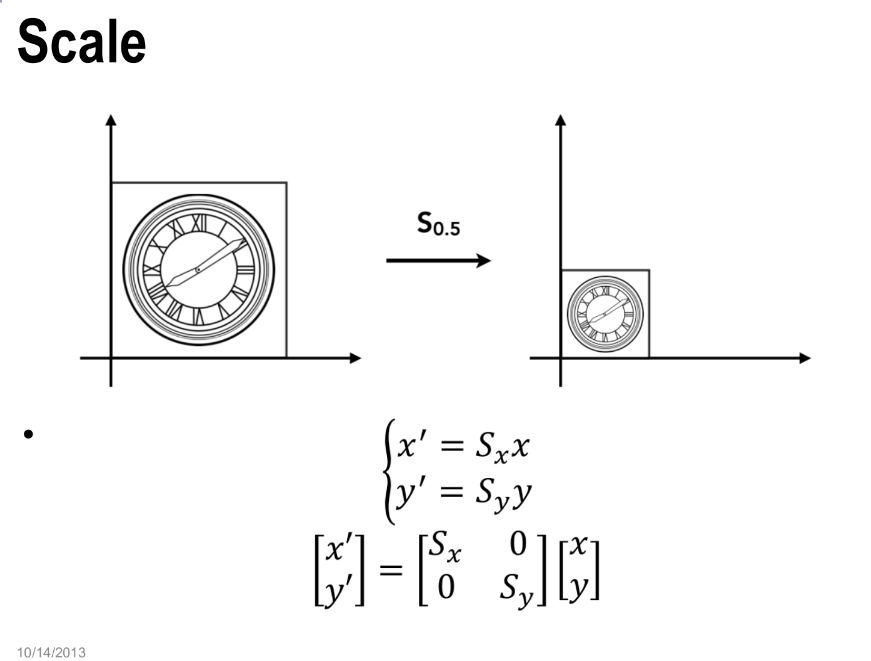
* Translation (phép dịch chuyển)
* Rotation (phép quay)
* Scale (phép scale)
* Projection (Phép phóng)
* Look at (Thiết lập điểm nhìn)

### 3.4.1 Translation

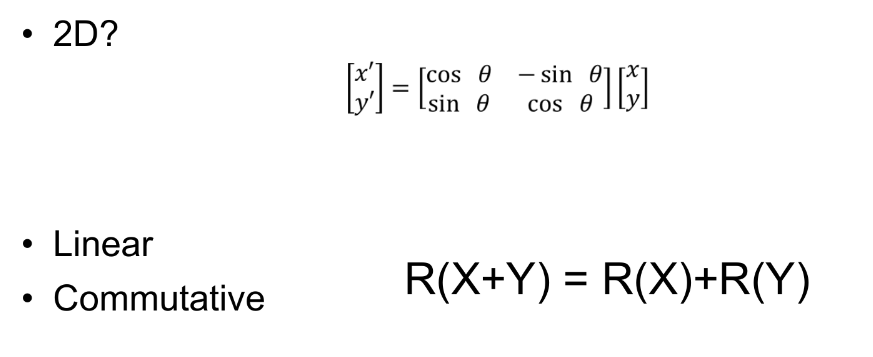


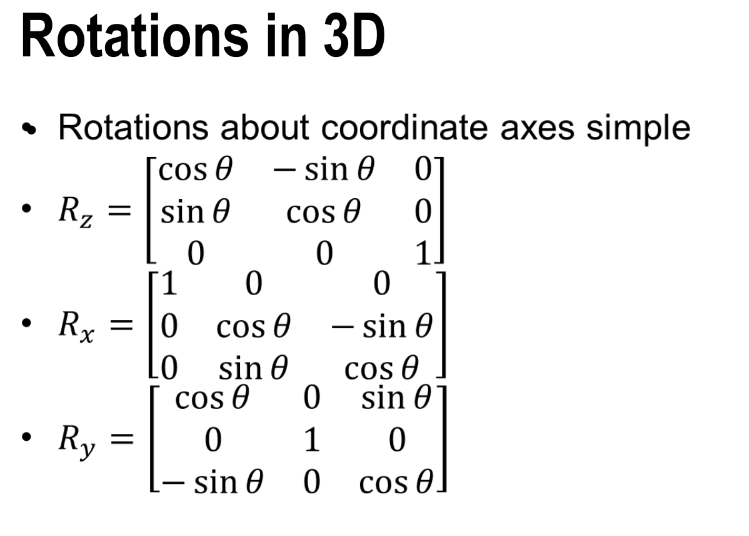
### 3.4.2 Scaling

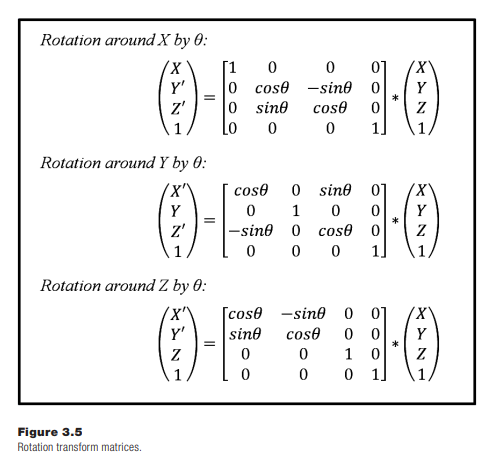




### 3.4.3 Rotation





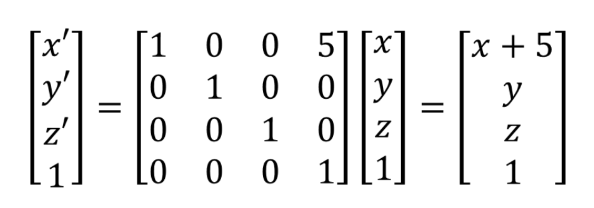


### 3.4.4 Homogeneous Coordinates

- Thêm chiều thứ 4 (w = 1)

- Ma trận 4x4 rất phổ biến trong đồ họa máy tính

- Hàng cuối cùng luôn là 0 0 0 1

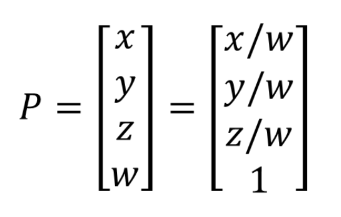


* Cách biểu diễn tọa độ Homogenous:

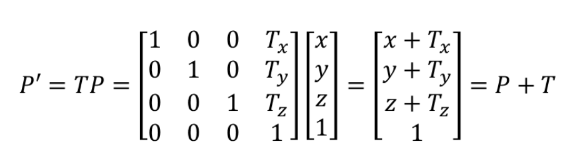
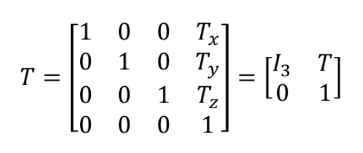
+ Chia cho tọa độ thứ 4 (w) để thu được điểm (không đồng nhất).

+ Phép nhân với w > 0, không ảnh hưởng.

+ Giả sử w >= 0. Với w > 0, là điểm hữu hạn thông thường. Với w = 0, là điểm tại vô cực(được sử dụng cho vector để dường việc tịnh tiến).

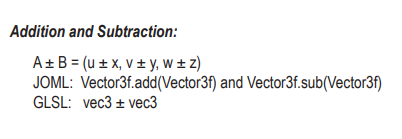


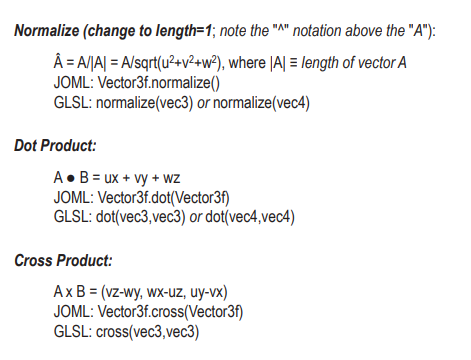
* Các biến đổi sử dụng Homogenous:



## 3.5. Vector

Các phép biến đổi vector: công, trừ vector, chuẩn hóa vector, tích vô hướng(Dot product), tích có hướng(Cross product).





## 3.6. LOCAL AND WORLD SPACE

**Local Space (Hệ tọa độ cục bộ)**

* **Local Space**, hay còn gọi là **Object Space**, là hệ tọa độ gắn liền với một đối tượng cụ thể. Trong hệ tọa độ này, tất cả các phép biến đổi và vị trí của các điểm đều được định nghĩa tương đối so với gốc tọa độ của chính đối tượng đó.
* **Gốc tọa độ (0,0,0)** trong Local Space thường nằm ở trung tâm của đối tượng hoặc một điểm đã định trước trên đối tượng.
* Khi làm việc với Local Space, các phép biến đổi như quay, dịch chuyển, và tỷ lệ được thực hiện trên đối tượng mà không phụ thuộc vào vị trí của nó trong thế giới.

Ví dụ:

* Một mô hình ô tô có các bánh xe. Trong Local Space của bánh xe, gốc tọa độ có thể nằm ở trục của bánh xe, và các điểm trên bánh xe được xác định dựa trên khoảng cách từ trục này.

**World Space (Hệ tọa độ thế giới)**

* **World Space** là hệ tọa độ toàn cục, nơi tất cả các đối tượng trong cảnh 3D được định vị tương đối với một gốc tọa độ chung của toàn bộ thế giới (thường là 0,0,00,0,00,0,0 của cảnh).
* Tất cả các đối tượng trong cảnh đều được chuyển đổi từ Local Space sang World Space để xác định vị trí và hướng của chúng trong không gian 3D.
* World Space giúp xác định mối quan hệ không gian giữa các đối tượng khác nhau, chẳng hạn như khoảng cách giữa chúng hoặc hướng tương đối.

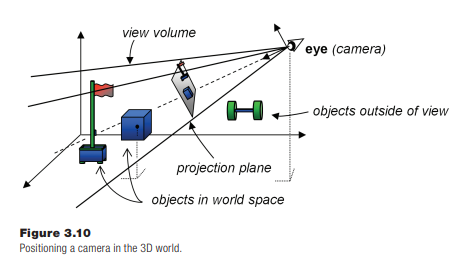
Ví dụ:

* Trong một trò chơi 3D, nếu ô tô được đặt ở vị trí (10,0,20)(10, 0, 20)(10,0,20) trong World Space, thì các tọa độ này xác định vị trí của ô tô trong không gian của toàn bộ thế giới.

## 3.7. EYE SPACE AND THE SYNTHETIC CAMERA

- **Eye Space** là hệ tọa độ mà trong đó các đối tượng được biểu diễn tương đối với camera, giúp đơn giản hóa các tính toán chiếu sáng và hiển thị.

- **Synthetic Camera** là mô hình camera ảo trong đồ họa máy tính, được sử dụng để biến cảnh 3D thành hình ảnh 2D, bao gồm các phép biến đổi nhìn và chiếu để mô phỏng góc nhìn của camera.



## 3.8. PROJECTION MATRICES

**Chức năng của Projection Matrices**

Khi hiển thị một cảnh 3D trên màn hình 2D, các đối tượng trong không gian 3D cần được chuyển đổi sao cho chúng có vẻ như có chiều sâu. Projection matrices thực hiện việc này bằng cách chiếu các điểm từ không gian tọa độ của camera (Eye Space) sang hệ tọa độ chuẩn hóa thiết bị (Normalized Device Coordinates - NDC), nơi các giá trị tọa độ nằm trong khoảng từ -1 đến 1 cho cả ba trục (x, y, z).

**Các loại Projection Matrices**

Có hai loại ma trận chiếu phổ biến:

1. **Perspective Projection Matrix (Ma trận chiếu phối cảnh)**
2. **Orthographic Projection Matrix (Ma trận chiếu trực giao)**

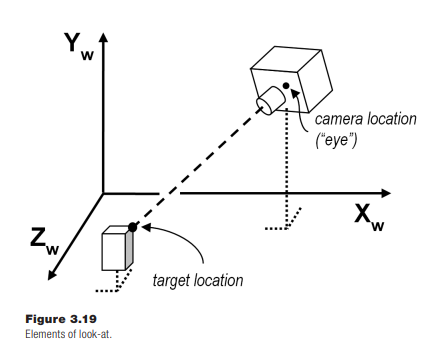
**1. Perspective Projection Matrix (Ma trận chiếu phối cảnh)**

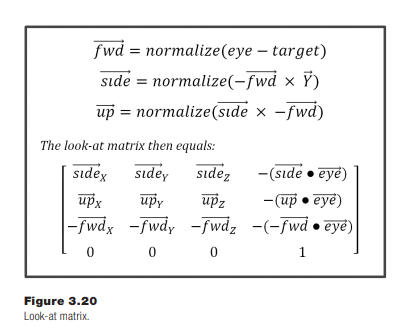
* **Mô tả:** Ma trận chiếu phối cảnh được sử dụng để tạo ra hiệu ứng phối cảnh, nơi các đối tượng xa hơn trong không gian 3D sẽ xuất hiện nhỏ hơn, trong khi các đối tượng gần hơn sẽ trông lớn hơn. Điều này tương tự như cách mà mắt người nhìn thấy thế giới, giúp tạo ra cảm giác về chiều sâu và khoảng cách.
* **Hoạt động:** Khi sử dụng phép chiếu phối cảnh, các tọa độ z sẽ được biến đổi sao cho các đối tượng xa có giá trị z lớn hơn (trong NDC) so với các đối tượng gần. Đồng thời, tọa độ x và y cũng được tỷ lệ theo giá trị z để tạo hiệu ứng phối cảnh.
* **Thông số quan trọng:** Góc nhìn (field of view - FOV), tỷ lệ khung hình (aspect ratio), và các mặt cắt gần xa (near và far clipping planes) giúp xác định phạm vi hiển thị của camera.

**2. Orthographic Projection Matrix (Ma trận chiếu trực giao)**

* **Mô tả:** Ma trận chiếu trực giao chiếu các đối tượng lên màn hình mà không có hiệu ứng phối cảnh, nghĩa là các đối tượng sẽ giữ nguyên kích thước bất kể chúng ở xa hay gần camera. Phép chiếu này phù hợp cho các ứng dụng như thiết kế CAD hoặc các trò chơi 2D.
* **Hoạt động:** Tất cả các điểm sẽ được chiếu theo các đường song song, và không có sự biến dạng tỷ lệ theo giá trị z như trong phép chiếu phối cảnh.
* **Thông số quan trọng:** Kích thước của vùng hiển thị (width và height), cùng với các mặt cắt gần xa (near và far clipping planes).

## 3.9. LOOK-AT MATRIX





## 3.10. GLSL FUNCTIONS FOR BUILDING MATRIX TRANSFORMS

## 3.11. SCENE GRAPH

**Scene Graphs** trong đồ họa máy tính là một cấu trúc dữ liệu phân cấp được sử dụng để tổ chức và quản lý các đối tượng trong một cảnh 3D. Nó giúp dễ dàng thực hiện và quản lý các phép biến đổi hình học (như dịch chuyển, quay, co giãn), cũng như các thuộc tính khác (như vật liệu, ánh sáng) của các đối tượng trong một cảnh phức tạp.

**Cấu trúc của Scene Graph**

Một **scene graph** thường được tổ chức dưới dạng một cây (tree), trong đó:

* **Nút gốc (root node):** Đại diện cho toàn bộ cảnh.
* **Các nút con (child nodes):** Đại diện cho các đối tượng hoặc nhóm đối tượng trong cảnh. Một nút con có thể chứa các nút con khác, tạo thành một cấu trúc phân cấp.
* **Mỗi nút trong đồ thị có thể đại diện cho:**
  + Một đối tượng hình học (hình khối, nhân vật, v.v.)
  + Các phép biến đổi (dịch chuyển, quay, co giãn)
  + Các thuộc tính như ánh sáng, vật liệu, hoặc texture

**Cách hoạt động của Scene Graph**

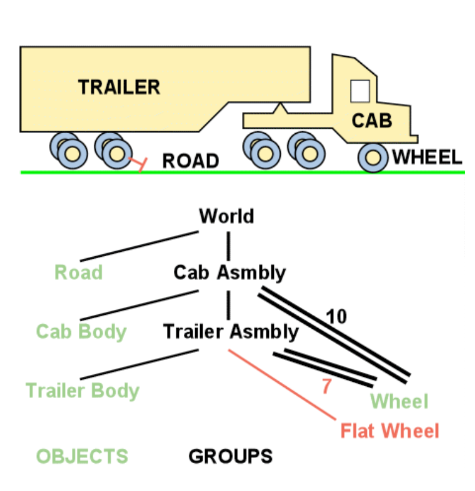
Các phép biến đổi và thuộc tính được áp dụng theo thứ tự từ nút gốc xuống các nút con. Điều này có nghĩa là:

1. **Kế thừa phép biến đổi:** Các nút con sẽ kế thừa các phép biến đổi từ nút cha của chúng. Ví dụ, nếu một đối tượng cha bị dịch chuyển, các đối tượng con của nó cũng sẽ dịch chuyển theo.
2. **Kết hợp phép biến đổi:** Các phép biến đổi có thể được kết hợp bằng cách nhân các ma trận biến đổi. Ví dụ, nếu một nút cha có phép quay và nút con có phép dịch chuyển, ma trận biến đổi kết hợp sẽ thực hiện cả hai phép biến đổi.
3. **Quản lý thuộc tính dễ dàng:** Scene graph cho phép dễ dàng thay đổi thuộc tính (như vật liệu hoặc ánh sáng) của toàn bộ nhánh của cây bằng cách thay đổi giá trị tại nút cha.

**Ví dụ về Scene Graph**

Giả sử bạn muốn tạo ra một mô hình ô tô với các bộ phận như thân xe, bánh xe, và cửa:

* **Nút gốc (ô tô):** Đại diện cho toàn bộ chiếc ô tô.
* **Các nút con (thân xe, bánh xe, cửa):** Mỗi nút con đại diện cho một phần của chiếc ô tô.
* **Phép biến đổi tại nút thân xe:** Nếu thân xe quay hoặc dịch chuyển, các bánh xe và cửa sẽ quay hoặc dịch chuyển theo.
* **Phép biến đổi riêng cho bánh xe:** Bánh xe có thể quay xung quanh trục của chúng mà không ảnh hưởng đến các bộ phận khác của ô tô.



# Chapter 4: MANAGING 3D GRAPHICS DATA

## 4.1. BUFFERS AND VERTEX ATTRIBUTES

Để 1 đối tượng được vẽ, các đỉnh của nó phải được gửi tới vertex shader. Các đỉnh thường được gửi bằng cách để chúng vào *buffer(bộ đệm)* bên phía Java và kết hợp buffer đó với 1 vertex attribute được khai báo trong shader. Có vài bước để hoàn thành việc đó, một vài thì chỉ cần làm 1 lần, 1 vài nếu cảnh đó chuyển động thì phải hoàn thành mỗi frame:

* Làm 1 lần – điển hình trong init():

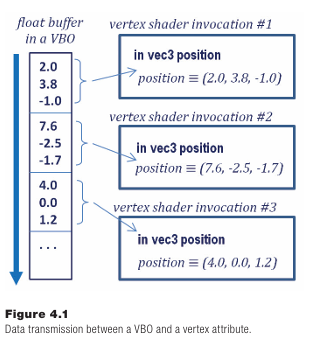
1. Tạo buffer.
2. Sao chép các đỉnh vào trong buffer.

* Làm mỗi frame, điển hình trong reshape():

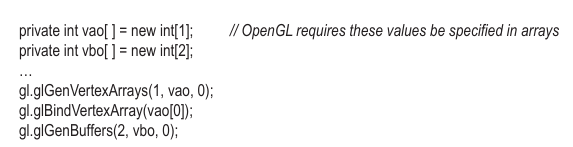
1. Tích hợp các buffer chứa các đỉnh.
2. Kết hợp các buffer với 1 vertex attribute
3. Tích hợp vertex attribute.
4. Sử dụng glDrawArrays(…) để vẽ đối tượng.

Những buffer được tạo tất cả trong 1 lần khi khởi chạy chương trình, trong init() hoặc trong hàm được gọi bởi init(). Trong OpenGL, 1 buffer được chứa trong 1 Vertex Buffer Object VBO. Một cảnh có thể cần nhiều VBO, bởi nó được tùy chỉnh để tạo và thực hiện chúng trong init(), thế nên chúng cần thiết khi chương trình của bạn cần vẽ 1 hay nhiều lần.

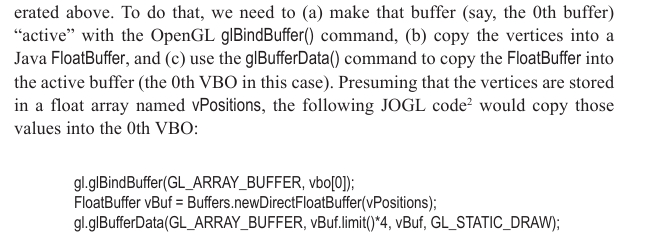
Một Buffer tương tác với 1 vertex attribute bởi 1 hướng xác định. Khi glDrawArrays() được thực thi, dữ liệu trong buffer bắt đầu chạy, tuần tự đi qua shader. Như đã nêu ở chương 2, vertex shader chỉ xử lý 1 lần 1 vector. 1 vector trong môi trường 3D cần 3 giá trị, thế nên thích hợp nhất là vec3. Sau đó mỗi 3 cặp giá trị trong buffer, shader sẽ thực thi:

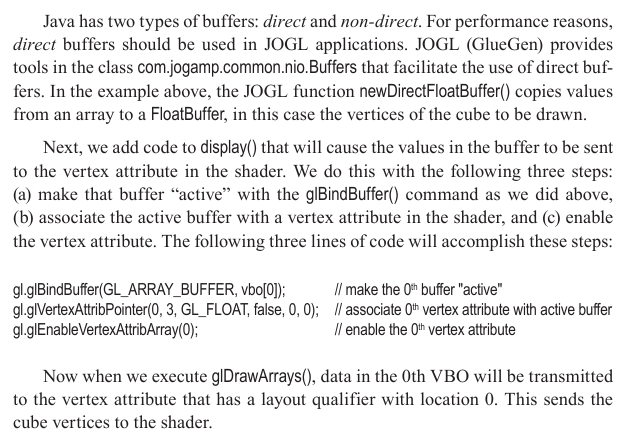


VAO(Vertex Array Object) được tạo ra nhầm cung cấp 1 hướng tổ chức các buffer và làm chúng dễ tích hợp hơn. OpenGL cần ít nhất 1 VAO được tạo.



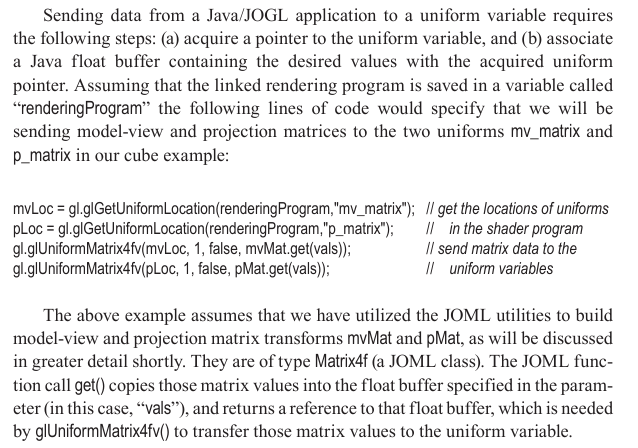
* glGenVertexArrays() và glGenBuffers(): tạo VAOs và VBOs, với tham số đầu là số vao và vbo.
* glBindVertexArrays(): xác định VAO nào được active.





## 4.2.UNIFORM VARIABLES

Uniform variables được khai báo trong 1 shader bằng cách sử dụng từ khóa “uniform”



## 4.3. INTERPOLATION OF VERTEX ATTRIBUTES

 **Xử lý thuộc tính đỉnh (vertex attributes) và biến thống nhất (uniform variables):**

* Các thuộc tính đỉnh được nội suy (interpolated) trong quá trình rasterization (phân mảnh), giúp chuyển đổi các đỉnh thành các đoạn nhỏ hơn gọi là "fragments" để tạo ra các điểm ảnh (pixels) trên màn hình. Quá trình này giúp các giá trị thuộc tính đỉnh được nội suy tuyến tính giữa các đỉnh để các pixel được hiển thị kết nối liền mạch với các bề mặt mô hình.
* Ngược lại, các biến thống nhất hoạt động như hằng số đã được khởi tạo và không thay đổi trong suốt quá trình xử lý của shader đỉnh, nghĩa là chúng giữ nguyên giá trị đối với mọi đỉnh. Không có sự nội suy cho các biến thống nhất; giá trị của chúng luôn không đổi, bất kể số lượng đỉnh.

 **Quá trình nội suy của thuộc tính đỉnh trong rasterizer:**

* Quá trình này rất hữu ích trong nhiều trường hợp, chẳng hạn như nội suy màu sắc, tọa độ kết cấu (texture coordinates), và các vector pháp tuyến bề mặt. Tất cả các giá trị được gửi qua bộ đệm đến các thuộc tính đỉnh sẽ được nội suy thêm trong pipeline.

 **Khai báo thuộc tính đỉnh trong shader:**

* Trong shader đỉnh, các thuộc tính đỉnh thường được khai báo với từ khóa "in" để nhận các giá trị từ bộ đệm. Tuy nhiên, chúng cũng có thể được khai báo là "out" để gửi giá trị đến các giai đoạn tiếp theo trong pipeline. Ví dụ, khai báo out vec4 color; nghĩa là thuộc tính đỉnh "color" được gửi ra ngoài dưới dạng vec4.
* Riêng với vị trí đỉnh (vertex position), OpenGL có sẵn biến gl\_Position để đảm nhận việc này, và trong shader đỉnh, các phép biến đổi ma trận sẽ được áp dụng để tính toán và gán giá trị cho gl\_Position.

 **Quá trình rasterization:**

* Khi sử dụng lệnh glDrawArrays() với tham số GL\_TRIANGLES, quá trình rasterization diễn ra cho từng tam giác. Nội suy bắt đầu dọc theo các đường nối các đỉnh với độ chính xác phụ thuộc vào mật độ điểm ảnh trên màn hình. Sau đó, các điểm ảnh bên trong tam giác được nội suy dọc theo các đường ngang nối các điểm biên.

Tóm lại, điểm khác biệt lớn nhất giữa thuộc tính đỉnh và biến thống nhất là thuộc tính đỉnh có thể được nội suy để tạo ra sự chuyển tiếp mượt mà giữa các giá trị, còn biến thống nhất thì không thay đổi trong suốt quá trình xử lý.

## 4.4. MODEL-VIEW AND PERSPECTIVE MATRICES

1. **Ba loại ma trận cần sử dụng:**
   * **Ma trận Mô hình (Model matrix):** Định vị và định hướng đối tượng trong không gian tọa độ thế giới. Mỗi mô hình có một ma trận mô hình riêng, và ma trận này cần được cập nhật liên tục nếu mô hình di chuyển.
   * **Ma trận Nhìn (View matrix):** Dịch chuyển và xoay các mô hình trong thế giới để mô phỏng hiệu ứng của một camera ở vị trí mong muốn. Camera của OpenGL luôn ở vị trí (0,0,0) và hướng về trục Z âm. Để tạo ra hiệu ứng di chuyển camera, cần di chuyển các đối tượng theo hướng ngược lại (ví dụ, nếu muốn camera di chuyển sang phải thì cần di chuyển các đối tượng sang trái).
   * **Ma trận Phối cảnh (Perspective matrix):** Tạo hiệu ứng 3D theo hình dạng frustum (khối chóp cụt) mong muốn, như đã mô tả trong các chương trước.
2. **Khi nào cần tính toán các ma trận này:**
   * Các ma trận không thay đổi có thể được tạo ra trong hàm init(), trong khi các ma trận thay đổi cần được tạo trong hàm display() để cập nhật cho mỗi khung hình (frame).
   * Nếu mô hình được hoạt hình hóa và camera có thể di chuyển:
     + Mỗi khung hình cần tạo ra ma trận mô hình cho từng mô hình.
     + Ma trận nhìn được tạo một lần mỗi khung hình và được áp dụng cho tất cả các đối tượng trong khung hình đó.
     + Ma trận phối cảnh được tạo một lần trong hàm init() dựa trên kích thước cửa sổ và các tham số frustum, và chỉ cần thay đổi nếu cửa sổ được thay đổi kích thước.
3. **Cách xây dựng các ma trận trong hàm display():**
   * **Bước 1:** Tạo ma trận nhìn dựa trên vị trí và hướng của camera mong muốn.
   * **Bước 2:** Với mỗi mô hình, thực hiện các bước sau:
     + Tạo ma trận mô hình dựa trên vị trí và hướng của mô hình.
     + Kết hợp ma trận mô hình và ma trận nhìn thành một ma trận "MV" duy nhất.
     + Gửi ma trận MV và ma trận phối cảnh đến các biến thống nhất trong shader.
4. **Lợi ích của việc kết hợp ma trận mô hình và ma trận nhìn:**
   * Không nhất thiết phải kết hợp hai ma trận này, nhưng việc này mang lại một số lợi ích về hiệu suất, đặc biệt khi mô hình phức tạp có hàng trăm hoặc hàng nghìn đỉnh. Kết hợp ma trận mô hình và ma trận nhìn trước khi gửi đến shader giúp giảm số lượng phép tính cần thiết trong shader.
   * Việc giữ ma trận phối cảnh riêng biệt sẽ cần thiết cho các hiệu ứng chiếu sáng sau này.

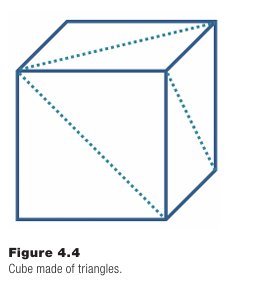
Tóm lại, quá trình này giúp định vị và hiển thị các đối tượng 3D trong không gian, đồng thời tối ưu hóa hiệu suất bằng cách quản lý hợp lý các ma trận biến đổi.

## 4.5. OUR FIRST 3D PROGRAM—A 3D CUBE

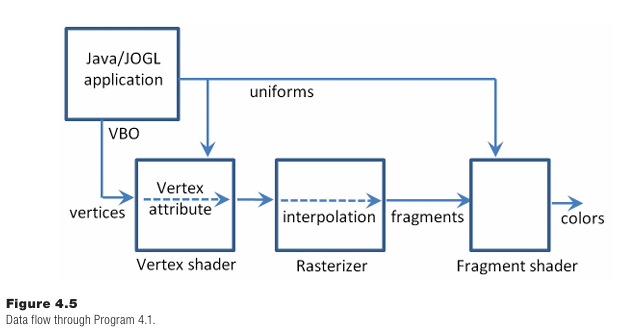
- File code trong thư mục src/book/

- Trong hàm setUpVertices():

+ Khai báo 1 mảng chứa 36 đỉnh. Tại sao khối hộp lại có 36 đỉnh? Tại vì mỗi mặt khối hộp được tạo từ được tạo từ 2 tam giác, tổng cộng cần là 12 tam giác.



* Mà mỗi tam giác được xác định bởi 3 đỉnh, tổng cộng là 36 điểm. Mỗi điểm có 3 giá trị x,y,z, tổng cộng là 108 giá trị trong mảng. Điều đó đúng nếu mỗi điểm tham gia vào nhiều tam giác, nhưng chúng ta vẫn xác định mỗi đỉnh riêng biệt vì chúng ta đang gửi mỗi đỉnh xuống pipeline một cách riêng biệt.
* Khối hộp được xác định trong 1 tọa độ, với (0,0,0) là tâm ở giữa, và với các góc trong khoảng từ -1.0 đến 1.0 trong 3 trục x, y, z. Phần còn lại của hàm là thiết lập VAO và VBOs, và load các đỉnh vào vbo buffer thứ 0.
* Ghi nhớ là hàm init() hoàn thành nhiệm vụ cần trong 1 lần: đọc shader và xây dựng chương trình render và load điểm vào buffer. Ghi nhớ nó cũng đặt vị trí khối hộp và camera trong thế giới; sau đó chúng ta se làm chuyển động khối hộp và cũng sẽ thấy làm cách nào để di chuyển camera xung quanh, tại nơi điểm chúng ta có thể loại bỏ vị trí cứng này.
* Bây giờ hãy nhìn vào hàm display(), gọi lại display() có thể gọi lặp lại và đánh giá cái mà được gọi lại trong frame rate. Chuyển động hoạt động liên tục bằng các vẽ và vẽ lại cảnh, hoặc khung hình, rất nhanh. Nó thường xuyên cần thiết để xóa chiều sâu của buffer trước khi render 1 khung hình, vì thế mặt ẩn sẽ diễn ra. Dùng glClear(GL\_DEPTH\_BUFFER\_BIT) để xóa chiều sâu buffer.
* Display() tích hợp shader bằng cách gọi glUseProgram() để cài đặt GLSL vào GPU. Gọi lại nó chạy lại shader program, nhưng nó sẽ có thể cho phép xếp vào hàng đợi OpenGL gọi để xác định vertex attribute của shader and uniform locations. Display() lấy uniform variable locations; xây dựng ma trận perspective, view, model. Liên kết ma trận view và ma trận model vào 1 ma trận mv; và chuyển perspective và ma trận mv cho corresponding uniforms.
* Display() cho phép buffer chứa điểm khối hộp và gắn chúng vào vertex attribute 0 để chuẩn bị gửi đến điểm để shader.
* Display() sẽ vẽ model bằng cách gọi glDrawArrays(), cụ thể là model cụ thể gồm các tam giác và tổng cộng 36 điểm.
* Cuối cùng xác định các shader. Vertex shader có vị trí vertex attribute. Bởi vì vị trí được xác định là 0 vì có thể tham chiếu dễ dàng bằng cách sử dụng 0 trong tham số đầu tiên của hàm glVertexAttribPointer() và trong glEnableVertexAttribArray(). Ghi nhớ vị trí vertex attribute được xác đinh bởi vec3, và nó cần thiết để chuyển sang 1 vec4 để thuận tiện với ma trận 4x4 bằng cách vec4(position, 1.0).
* Sự đa hình của vertex shader cho phép ma trận chuyển sang đỉnh, chuyển nó sang không gian camera. Những giá trị đó được đặt vào built-in OpenGL biến output gl\_Position và sau đó tiếp tục đi qua pipeline và nội suy bởi rasterizer.
* Tọa độ pixel nội suy sau đó được gửi tới fragment shader. Giống như vertex shader thì fragment shader cũng xử lý từng pixel một.



## 4.6. RENDERING MULTIPLE CONPIES OF AN OBJECT

### 4.6.1. Instancing

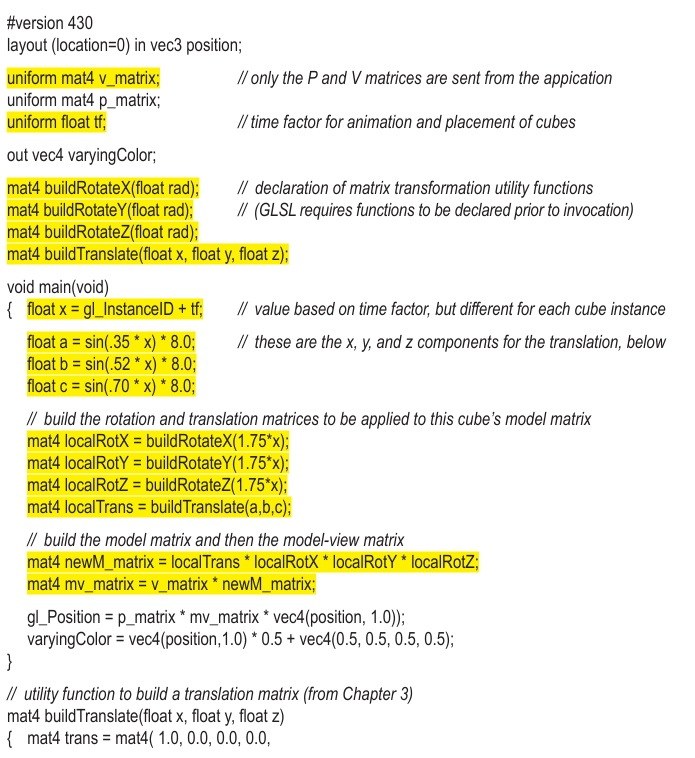
- Sử dụng hàm sau thay thế glDrawArrays() để vẽ đối tượng nhầm tối ưu performance:

glDrawArraysInstanced()

* Khi sử dụng instancing thì vertex shader có thể truy cập biến built-in, get\_InstanceID, 1 số nguyên trong glsl:



* Khi sử dụng instancing, thì vertex shader sẽ truy cập vào 1 biến built-in, gl\_InstanceID, 1 số int tham chiếu đến số instance của object hiện tại đang xử lý.



* Sử dụng glDrawArraysInstanced(GL\_TRIANGLES, 0, 36, 24) để vẽ

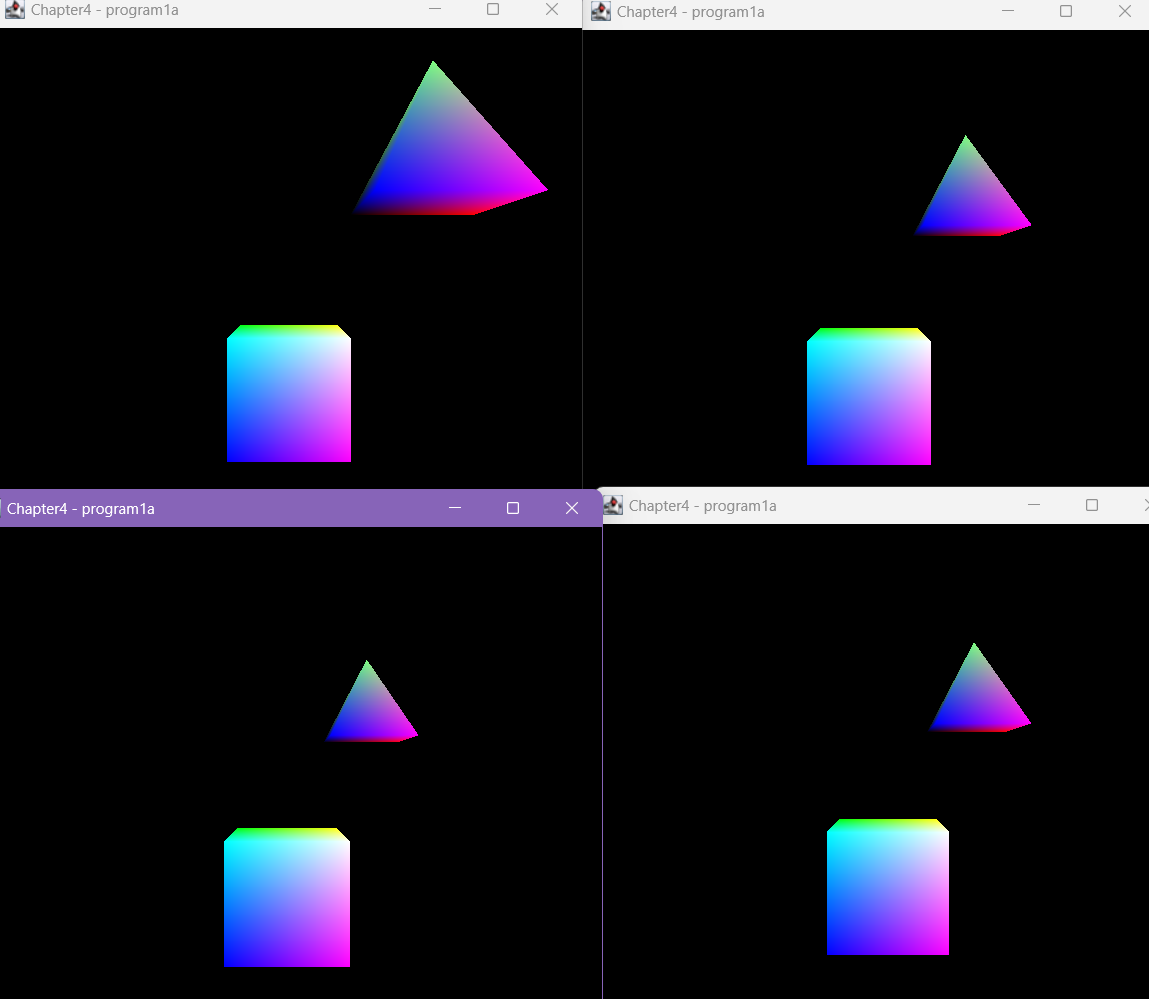
## 4.7. RENDERING MULTIPLE DIFFERENT MODELS IN SCENE

Để dựng nhiều hơn 1 model trong 1 khung cảnh, 1 cách tiếp cận đơn giản là sử dụng 1 separate buffer cho mỗi model. Mỗi model sẽ cần 1 ma trận model của nó, và như vậy 1 ma trận model-view sẽ được tạo cho mỗi model khi chúng ta dựng. Sẽ có nhiều thứ cần làm để chia gọi lại glDrawArrays() mỗi model. Có nhiều thay đổi trong init() và trong display().

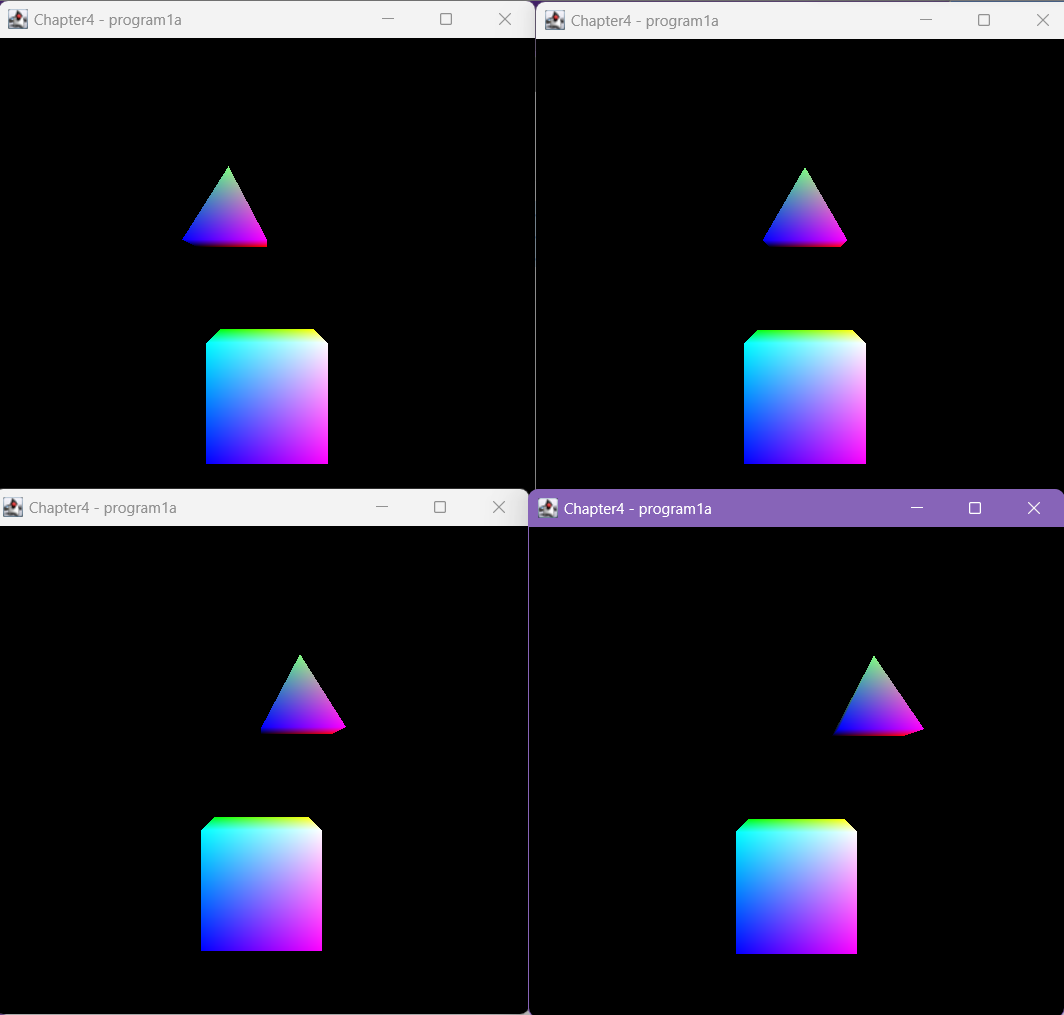
Một hướng khác là chúng ta sẽ cần shader khác hoặc 1 chương trình render khác cho mỗi đối tượng chúng ta muốn vẽ. Cách làm là chúng ta có thể tái sử dụng cùng 1 vertex and fragment shaders, và chỉ cần điều chỉnh chương trình Java/JOGL để gửi mỗi model xuống pipeline khi display() được gọi.

Tiếp tục bằng cách thêm 1 hình chóp, bởi cảnh của chúng ta gồm 1 khối họp và 1 khói chóp. Note: 1 khối chóp được tạo bởi 6 tam giác tổng cộng 6x3=18 đỉnh.

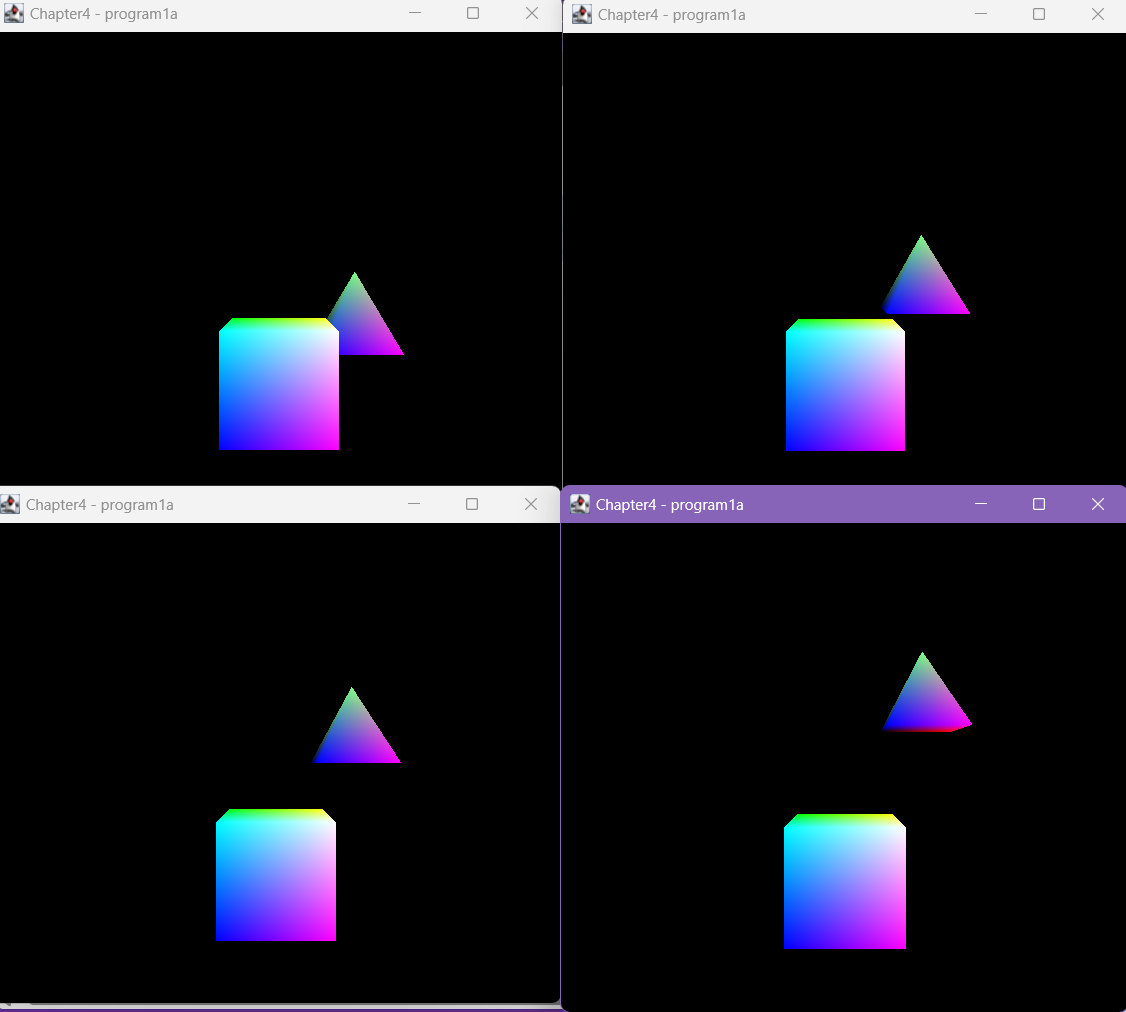
\*\*\* LocZ chỉnh độ xa gần màn hình:



\*\*\* LocX: chiều ngang



\*\*\*LocY: trục thẳng



* Ma trận view được đặt trên cùng hàm display() và để sử dụng trong ma trận model-view cho cả khối hộp và khối chóp.

## 4.8 MATRIX STACKS

Khi chúng ta dựng thì cần thiết lập 1 tập đơn các điểm. Tuy nhiên để xây dựng linh hoạt model bằng cách tập hơp các model đơn giản nhỏ hơn. Một đối tượng được xây dựng như thế gọi là 1 hierarchical(thứ bậc) model.

1 matrix stack là 1 hàng chờ của các ma trận chuyển đổi. Matrix stack giúp dễ dàng tạo và quản lý các đối tượng và cảnh thứ bậc 1 cách tùy ý, nơi những chuyển đổi có thể dựng trong lúc cái khác chuyển đổi. Nhiều ma trận model, view, model-view sễ có thể cần trong 1 cảnh tùy biến có thể bị thay thế bởi 1 instance của Matrixf4Stack.

Một vài hàm quan trọng của Matrix4fStack:

* pushMatrix(): tạo 1 bản copy của ma trận trên cùng và đẩy bản copy đó vào stack.
* popMatrix(): xóa và trả về ma trận trên cùng.
* get(v): copy giá trị ma trận cao nhất vào buffer v và trả về 1 tham chiếu đến buffer (ma trận đó không bị xóa khỏi stack).
* Rotate(d, x, y, z)
* Scale(x, y, z) thiết lập hướng của ma trận trên cùng của stack
* Translate(x, y, z)

Matrix4fStack được thiết kế như 1 subclass của Matrix4f, vì vậy thừa hưởng các hàm có trong Matrix4f. Nếu 1 trong các hàm trong Matrix4fStack, nó sẽ áp dụng lên ma trận trên cùng của stack.

Ma trận đầu tiên được push vào stack thường là ma trận VIEW. Những ma trận phía trên nó là ma trận model-view với mức độ phức tạp tăng dần; nghĩa là, chúng có nhiều biến đổi mô hình hơn được áp dụng vào. Các biến đổi này có thể được áp dụng trực tiếp hoặc bằng cách gộp các ma trận khác vào trước.

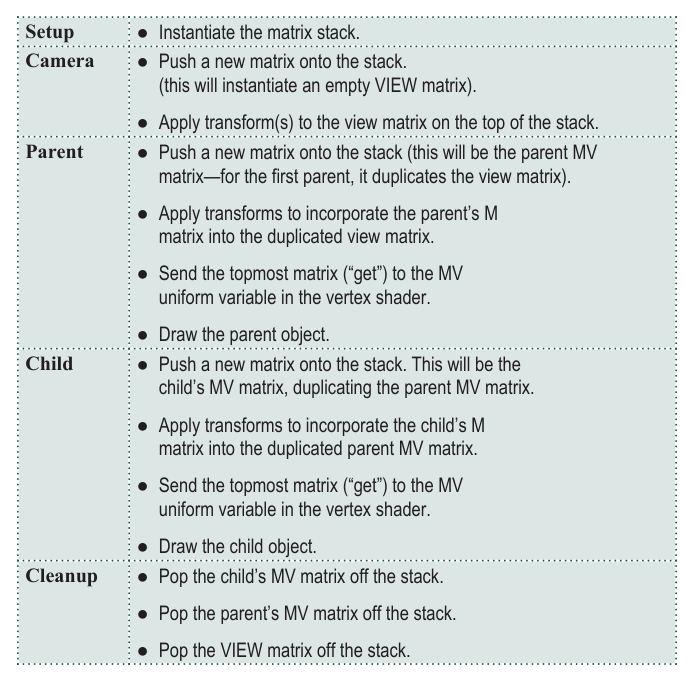
Trong ví dụ về hệ trái đất, ma trận nằm ngay trên ma trận VIEW sẽ là ma trận MV của mặt trời. Ma trận nằm trên ma trận đó sẽ là ma trận MV của Trái đất, bao gồm một bản sao của ma trận MV của mặt trời với các biến đổi mô hình của Trái đất được áp dụng vào. Nói cách khác, ma trận MV của Trái đất được tạo ra bằng cách kết hợp các biến đổi của trái đất này vào các biến đổi của mặt trời. Tương tự, ma trận MV của mặt trăng nằm trên ma trận MV của trái đất và được tạo ra bằng cách áp dụng các biến đổi của ma trận mô hình mặt trăng lên ma trận MV của trái đất nằm ngay bên dưới nó.

Sau khi hiển thị mặt trăng, một "mặt trăng" thứ hai có thể được hiển thị bằng cách “pop” ma trận của mặt trăng đầu tiên ra khỏi ngăn xếp (khôi phục đỉnh ngăn xếp về ma trận model-view của trái đất) và sau đó lặp lại quá trình này cho mặt trăng thứ hai.

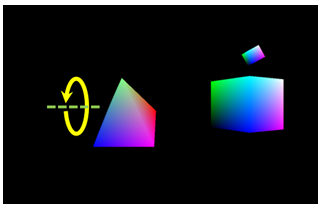
Cách tiếp cận cơ bản như sau:

1. Khi một đối tượng mới được giới thiệu so với một đối tượng cha, thực hiện một lệnh “pushMatrix()”.
2. Áp dụng các biến đổi của đối tượng mới; tức là nhân một biến đổi vào ma trận ở đỉnh ngăn xếp.
3. Khi một đối tượng hoặc tiểu đối tượng được vẽ xong, gọi lệnh “popMatrix()” để loại bỏ ma trận model-view của nó khỏi đỉnh của ngăn xếp.

Trong các chương sau, chúng ta sẽ học cách tạo ra các hình cầu và làm cho chúng trông giống như các trái đất và mặt trăng. Hiện tại, để đơn giản hóa, chúng ta sẽ xây dựng một "hệ trái đất" bằng cách sử dụng kim tự tháp và một vài khối lập phương.



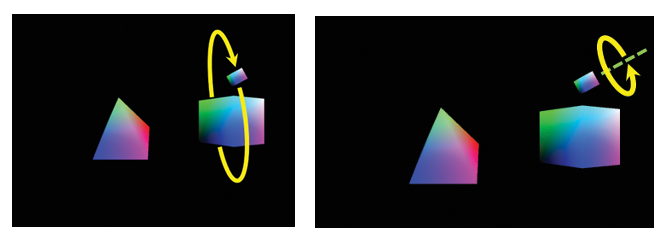
Lưu ý rằng sự quay của kim tự tháp (“mặt trời”) quanh trục của nó nằm trong hệ tọa độ cục bộ của riêng nó và không nên ảnh hưởng đến các “đối tượng con” (trong trường hợp này là trái đất và mặt trăng). Do đó, sự quay của mặt trời (như hình minh họa bên dưới) được đưa vào ngăn xếp, nhưng sau khi vẽ xong mặt trời, nó cần được loại bỏ (pop) khỏi ngăn xếp.



Sự quay xung quanh mặt trời của khối lập phương lớn (trái đất) (hình bên trái, bên dưới) sẽ ảnh hưởng đến chuyển động của mặt trăng, do đó nó được đưa vào ngăn xếp và vẫn giữ nguyên khi vẽ mặt trăng. Ngược lại, sự quay quanh trục của trái đất (hình bên phải, bên dưới) là chuyển động cục bộ và không ảnh hưởng đến mặt trăng, vì vậy nó sẽ được loại bỏ (pop) khỏi ngăn xếp trước khi vẽ mặt trăng.



Tương tự, chúng ta sẽ đưa các biến đổi vào ngăn xếp cho các chuyển động quay của mặt trăng (quay xung quanh trái đất và quanh trục của nó), như được chỉ ra trong các hình ảnh sau.



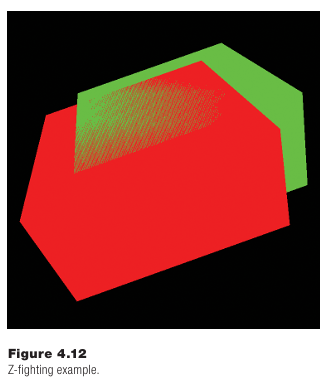
Dưới đây là trình tự các bước cho “trái đất”:

* pushMatrix() — Đây sẽ là phần của ma trận MV của trái đất mà cũng sẽ ảnh hưởng đến các đối tượng con.
* translate(...) — Thao tác này sẽ thêm chuyển động của trái đất xung quanh mặt trời vào ma trận MV của trái đất. Trong ví dụ này, chúng ta sử dụng lượng giác để tính toán chuyển động của trái đất như một phép tịnh tiến.
* pushMatrix() — Đây sẽ là ma trận MV hoàn chỉnh của trái đất, bao gồm cả sự quay quanh trục của nó.
* rotate(...) — Thao tác này sẽ thêm vào sự quay quanh trục của trái đất (sau đó sẽ được loại bỏ (pop) và không ảnh hưởng đến các đối tượng con).
* get() — Thao tác này sẽ lấy ma trận MV và gửi nó đến đồng phục MV (MV uniform).
* Vẽ trái đất.
* popMatrix() — Thao tác này sẽ loại bỏ ma trận MV của trái đất khỏi ngăn xếp, làm lộ ra bản sao trước đó của ma trận MV của trái đất mà không bao gồm sự quay quanh trục của trái đất (để chỉ có phép tịnh tiến của trái đất mới ảnh hưởng đến mặt trăng).

## 4.9 COMBATING “Z\_FIGHTING” ARTIFACTS

Hãy nhớ rằng khi vẽ nhiều đối tượng, OpenGL sử dụng thuật toán Z-buffer (đã được trình bày trước đó trong Hình 2.14) để loại bỏ các bề mặt bị che khuất. Thông thường, thuật toán này xác định các bề mặt nào của đối tượng sẽ hiển thị và được vẽ trên màn hình, so với các bề mặt nằm sau các đối tượng khác và do đó không nên được vẽ, bằng cách chọn màu của một pixel là màu của đoạn (fragment) tương ứng gần nhất với camera.

Tuy nhiên, có thể có những trường hợp hai bề mặt của đối tượng trong một cảnh trùng nhau và nằm trên các mặt phẳng trùng khớp, khiến thuật toán Z-buffer gặp khó khăn trong việc xác định bề mặt nào trong số hai bề mặt đó nên được vẽ (vì không có bề mặt nào "gần nhất" với camera). Khi điều này xảy ra, các lỗi làm tròn số của số thực có thể khiến một số phần của bề mặt được vẽ sử dụng màu của một đối tượng, trong khi các phần khác sử dụng màu của đối tượng kia. Hiện tượng này được gọi là "Z-fighting" hoặc "depth-fighting", vì hiệu ứng này là kết quả của việc các đoạn vẽ “đấu tranh” với nhau để chiếm các mục nhập pixel tương ứng trong Z-buffer. Hình 4.12 minh họa một ví dụ về Z-fighting giữa hai hộp có các mặt trùng khớp (mặt trên) chồng lên nhau.



Các tình huống như vậy thường xảy ra khi tạo địa hình hoặc bóng đổ. Trong các trường hợp như vậy, thường có thể dự đoán được hiện tượng Z-fighting, và một cách phổ biến để khắc phục nó là di chuyển một đối tượng đi một chút để các bề mặt không còn đồng phẳng. Chúng ta sẽ xem một ví dụ về điều này ở Chương 8.

Z-fighting cũng có thể xảy ra do độ chính xác giới hạn của các giá trị trong bộ đệm chiều sâu (depth buffer). Đối với mỗi pixel được xử lý bởi thuật toán Z-buffer, độ chính xác của thông tin về chiều sâu bị giới hạn bởi số lượng bit có sẵn để lưu trữ trong bộ đệm chiều sâu. Khoảng cách càng lớn giữa các mặt phẳng cắt gần và xa được sử dụng để xây dựng ma trận phối cảnh, thì càng có khả năng các điểm của hai đối tượng với độ sâu thực tế tương tự nhau (nhưng không bằng nhau) sẽ được biểu diễn bằng cùng một giá trị số trong bộ đệm chiều sâu. Do đó, lập trình viên cần chọn các giá trị cho mặt phẳng cắt gần và xa sao cho khoảng cách giữa hai mặt phẳng này là nhỏ nhất, trong khi vẫn đảm bảo rằng tất cả các đối tượng quan trọng trong cảnh nằm trong khung nhìn (viewing frustum).

Cũng cần hiểu rằng, do ảnh hưởng của phép biến đổi phối cảnh, việc thay đổi giá trị của mặt phẳng cắt gần có thể tác động lớn hơn đến khả năng xuất hiện hiện tượng Z-fighting so với thay đổi tương tự ở mặt phẳng cắt xa. Vì vậy, tốt nhất là tránh chọn mặt phẳng cắt gần quá sát với mắt.

Trong các ví dụ trước trong sách này, chúng ta chỉ đơn giản sử dụng các giá trị 0.1 và 1000 (trong các lệnh gọi hàm perspective()) cho các mặt phẳng cắt gần và xa. Các giá trị này có thể cần phải điều chỉnh cho phù hợp với cảnh của bạn.

## 4.10 OTHER OPTIONS FOR PRIMITIVES

## 4.11 CODING FOR PERFORMANCE

### 4.11.1 Minimizing Dynamic Memory Allocation

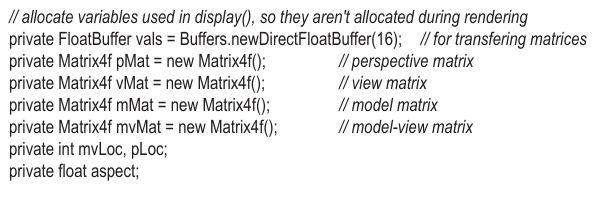
Module quan trọng trong các ứng dụng Java/JOGL của chúng ta, về mặt hiệu suất, chính là hàm display(). Đây là hàm được gọi lặp đi lặp lại trong bất kỳ hoạt ảnh hay quá trình hiển thị thời gian thực nào, vì vậy đây cũng là nơi (hoặc trong bất kỳ hàm nào mà nó gọi đến) mà chúng ta cần tối ưu hóa để đạt hiệu suất cao nhất.

Một cách quan trọng để giảm thiểu chi phí trong hàm display() là tránh bất kỳ thao tác nào yêu cầu cấp phát bộ nhớ. Các ví dụ rõ ràng về những việc cần tránh bao gồm:

* Khởi tạo đối tượng
* Khai báo biến
* Cấp phát bộ đệm

Chúng tôi khuyến khích người đọc xem lại từng chương trình mà chúng ta đã phát triển cho đến nay và lưu ý rằng mọi biến được sử dụng trong hàm display() (ngoại trừ biến gl kiểu GL4) đều đã được khai báo và cấp phát bộ nhớ trước khi hàm display() thực sự được gọi. Một ví dụ khác là Matrix4fStack, trong đó JOML yêu cầu cấp phát trước dung lượng tối đa của ngăn xếp, để toàn bộ ngăn xếp được cấp phát trước, tránh việc cấp phát không gian động khi thực hiện các thao tác “push” trong display().

Thực tế là chúng ta hiện đang giảm thiểu số lượng khai báo hoặc khởi tạo của bất kỳ loại nào xuất hiện trong display(). Chẳng hạn, trong Chương trình 4.1, có đoạn mã sau được đặt ở đầu chương trình.



Lưu ý rằng chúng ta đã cố ý đặt một chú thích ở đầu khối mã để chỉ ra rằng các biến này được cấp phát trước để sử dụng sau này trong hàm display() (mặc dù đến bây giờ chúng ta mới nêu rõ điều đó).

Cũng có các ví dụ tinh vi hơn. Chẳng hạn, các lệnh gọi hàm chuyển đổi dữ liệu từ loại này sang loại khác có thể trong một số trường hợp sẽ khởi tạo và trả về dữ liệu đã chuyển đổi. Vì vậy, điều quan trọng là cần hiểu rõ hành vi của bất kỳ hàm thư viện nào được gọi từ display().

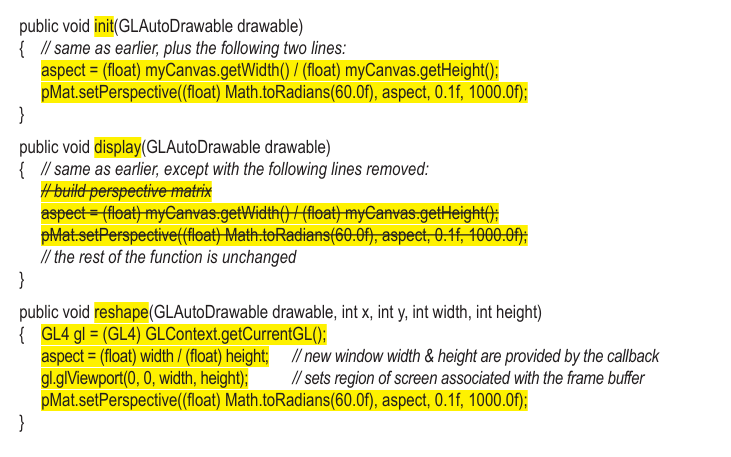
May mắn thay, thư viện toán học JOML mà chúng ta đang sử dụng đã được thiết kế cẩn thận để loại bỏ (hoặc ít nhất là giảm thiểu) việc cấp phát bộ nhớ khi gọi các hàm của nó, đồng thời vẫn cung cấp các chức năng cần thiết cho OpenGL. Hầu hết các hàm của JOML hoạt động trực tiếp trên đối tượng từ đó hàm được gọi hoặc trên một trong các tham số của nó (thay vì xây dựng một kết quả và trả về), do đó cho phép không gian lưu trữ kết quả có thể được cấp phát trước.

Một ví dụ về cách JOML được thiết kế khéo léo để tối ưu hóa hiệu suất theo cách này có thể thấy trong các lệnh gọi glUniformMatrix4fv(). Hãy cùng xem kỹ một lệnh gọi như vậy, lệnh chuyển ma trận model-view vào một biến uniform.



### 4.11.2 Pre-Computing the Perspective Matrix

May mắn thay, JOGL tự động gọi hàm reshape() bất cứ khi nào cửa sổ được thay đổi kích thước. Cho đến giờ, chúng ta đã để hàm reshape() trống — và bây giờ chúng ta đã có một mục đích sử dụng cho nó. Chúng ta chỉ cần di chuyển mã tính toán ma trận phối cảnh vào trong init(), và cũng sao chép mã này vào trong reshape().



### 4.11.3 Back-Face Culling

Một cách khác để cải thiện hiệu suất hiển thị là tận dụng khả năng của OpenGL trong việc loại bỏ mặt sau (back-face culling). Khi một mô hình 3D được "đóng" hoàn toàn, có nghĩa là bên trong của nó không bao giờ được nhìn thấy (như trong trường hợp của khối lập phương và hình chóp), thì những phần bề mặt ngoài có góc quay ra xa khỏi người xem sẽ luôn bị che khuất bởi một phần khác của cùng mô hình. Nghĩa là, các tam giác quay mặt ra khỏi người xem sẽ không thể nhìn thấy (chúng sẽ bị loại bỏ trong quá trình xóa bề mặt ẩn), vì vậy không có lý do gì để rasterize hoặc hiển thị chúng.

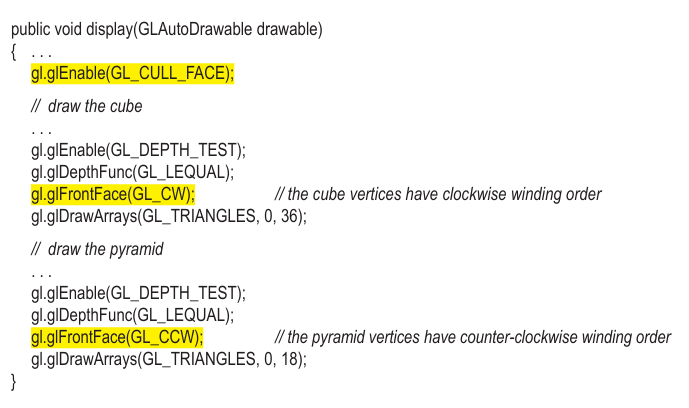
Chúng ta có thể yêu cầu OpenGL xác định và "loại bỏ" (không hiển thị) các tam giác quay mặt sau bằng lệnh glEnable(GL\_CULL\_FACE). Chúng ta cũng có thể tắt loại bỏ mặt bằng glDisable(GL\_CULL\_FACE). Mặc định, loại bỏ mặt bị tắt, nên nếu muốn OpenGL loại bỏ các tam giác quay mặt sau, bạn phải bật nó lên. Khi loại bỏ mặt được bật, mặc định các tam giác chỉ được hiển thị nếu chúng quay mặt trước. Cũng theo mặc định, một tam giác được coi là quay mặt trước nếu ba đỉnh của nó được xếp theo hướng ngược chiều kim đồng hồ (dựa trên thứ tự mà chúng được xác định trong bộ đệm) khi nhìn từ camera của OpenGL. Các tam giác có đỉnh xếp theo chiều kim đồng hồ (khi nhìn từ camera OpenGL) là quay mặt sau và sẽ không được hiển thị.

Định nghĩa quay mặt trước theo chiều ngược kim đồng hồ này đôi khi được gọi là *winding order* và có thể được đặt rõ ràng bằng cách gọi hàm glFrontFace(GL\_CCW) cho ngược chiều kim đồng hồ (mặc định) hoặc glFrontFace(GL\_CW) cho chiều kim đồng hồ. Tương tự, có thể đặt rõ ràng là tam giác quay mặt trước hay mặt sau được hiển thị. Thực ra, với mục đích này, chúng ta sẽ chỉ định cái nào không được hiển thị — tức là cái nào bị “loại bỏ”. Chúng ta có thể chỉ định rằng các tam giác quay mặt sau bị loại bỏ (dù không cần thiết vì đó là mặc định) bằng cách gọi glCullFace(GL\_BACK). Ngoài ra, có thể chỉ định thay thế rằng các tam giác quay mặt trước bị loại bỏ, hoặc thậm chí là tất cả các tam giác đều bị loại bỏ, bằng cách thay thế tham số GL\_BACK bằng GL\_FRONT hoặc GL\_FRONT\_AND\_BACK tương ứng.

Như chúng ta sẽ thấy ở Chương 6, các mô hình 3D thường được thiết kế sao cho bề mặt ngoài được tạo thành từ các tam giác có cùng hướng quay—thường là ngược chiều kim đồng hồ—để nếu bật loại bỏ mặt thì mặc định phần bề mặt ngoài của mô hình hướng về camera sẽ được hiển thị. Vì mặc định OpenGL giả định hướng quay là ngược chiều kim đồng hồ, nếu một mô hình được thiết kế để hiển thị với hướng quay theo chiều kim đồng hồ, lập trình viên cần gọi gl\_FrontFace(GL\_CW) để điều chỉnh nếu loại bỏ mặt sau được bật.

Lưu ý rằng trong trường hợp GL\_TRIANGLE\_STRIP, hướng quay của mỗi tam giác thay đổi luân phiên. OpenGL xử lý việc này bằng cách “đảo chiều” thứ tự đỉnh khi tạo mỗi tam giác kế tiếp, như sau: 0-1-2, sau đó 2-1-3, 2-3-4, 4-3-5, 4-5-6, và cứ thế tiếp tục.

Loại bỏ mặt sau cải thiện hiệu suất bằng cách đảm bảo rằng OpenGL không tốn thời gian rasterize và hiển thị các bề mặt không được định sẵn để nhìn thấy. Hầu hết các ví dụ trong chương này có kích thước nhỏ, nên không có động lực lớn để bật loại bỏ mặt (ngoại trừ ví dụ trong Hình 4.9 với 100.000 khối lập phương hoạt hình được lặp lại, điều này có thể gây khó khăn về hiệu suất trên một số hệ thống). Trên thực tế, hầu hết các mô hình 3D thường là “đóng,” do đó, việc bật loại bỏ mặt sau là thông lệ phổ biến. Ví dụ, chúng ta có thể thêm loại bỏ mặt sau vào Chương trình 4.3 bằng cách sửa đổi hàm display() như sau:



Việc thiết lập đúng thứ tự winding là rất quan trọng khi sử dụng loại bỏ mặt sau (back-face culling). Một thiết lập sai, chẳng hạn chọn GL\_CW khi lẽ ra phải là GL\_CCW, có thể khiến nội thất của một đối tượng được hiển thị thay vì phần ngoại thất của nó, từ đó gây ra hiện tượng méo hình tương tự như khi ma trận phối cảnh không đúng.

Hiệu quả không phải là lý do duy nhất để sử dụng loại bỏ mặt. Trong các chương sau, chúng ta sẽ thấy những ứng dụng khác, chẳng hạn như trong những trường hợp mà chúng ta muốn nhìn thấy bên trong của một mô hình 3D hoặc khi sử dụng tính năng trong suốt.