**Lời Cảm Ơn**

Lời đầu tiên tôi xin bày tỏ lòng biết ơn chân thành tới các thầy cô giáo Trường Đại Học Bách Khoa – Đại Học Đà Nẵng nói chung và các thầy cô bộ môn công nghệ phần mềm nói riêng. Trong suốt thời gian tôi học tại trường, các thầy cô đã luôn tận tình dạy dỗ, chỉ bảo để tôi có được kết quả như ngày hôm nay.

Đặc biệt, tôi xin chân thành cảm ơn giảng viên, tiến sĩ Lê Thị Mỹ Hạnh. Cảm ơn cô vì những định hướng, nhận xét quý báu và động viên kịp thời để tôi hoàn thành khóa luận này. Tôi cũng cảm ơn Gameloft – Đà Nẵng đã tạo điều kiện cho tôi có môi trường làm việc trong quá trình hoàn thành khóa luận này.

Cuối cùng tôi xin bày tỏ lòng biết ơn sâu sắc tới gia đình, bạn bè, những người đã luôn động viên, giúp đỡ tôi cả về vật chất lẫn tinh thần trong suốt những năm tháng qua.

Đà nẵng, ngày …. Tháng … năm…

Sinh viên

**DANH MỤC CÁC TỪ VIẾT TẮT**

Bounding box Hình hộp bé nhất chứa trọn vẹn các đối tượng 3D

Mesh : Các đa giác dùng để mô tả hình dạng của vật thể trong không gian 3 chiều

Local space: không gian riêng của đối tượng 3D

Clip space : không gian tọa độ vertex được chiếu lên bởi camera

Cube map: là tập hợp 6 texture 2D được xếp lien tục trong 6 mặt của hình hộp

Shader : là tập các chỉ lệnh cho GPU được dùng để Render các hiệu ứng.

Texture: Các hình ảnh được dán lên các đối tượng 3D nhằm tăng độ chi tiết của đối tượng

Vertex : Đỉnh trong không gian 3D

Vertex program : chương trình shader xử lí các biến đổi với vertex

World space : không gian thế giới trong OpenGL

Fragment program : chương trình shader xử lí các biến đổi với mỗi pixel.

Vertex buffer : vùng đệm chứa các vertex.

NDC : Normalized Device Coordinates (Hệ trục tọa độ thiết bị)

NDK: Native Development Kit

JVM : Java Virtual Machine

JNI (Java Native Interface)

**CHƯƠNG 1 – CHƯƠNG MỞ ĐẦU**

* 1. **Lý do chọn đề tài**

Ngành công nghiệp phát triển game trên thế giới đang phát triển như vũ bão. Cách đây 20 năm, game rất đơn giản và chỉ một người hoặc một nhóm nhỏ người phát triển nhưng ngày nay, các game hiện đại với đồ họa và khả năng tương tác ấn tượng thường được phát triển bởi một đội ngũ đông đảo người thiết kế game, lập trình viên, nghệ sĩ… trong thời gian ròng rã từ một đến ba năm. Chính việc game ngày càng trở nên phức tạp như vậy nên hiện nay, các game thương mại không còn phát triển từ nguyên thủy nữa mà được phát triển lên từ game engine. Game engine hỗ trợ việc xây dựng game một cách nhanh chóng, đơn giản hơn đồng thời cung cấp khả năng tái sử dụng code cao do đó có thể phát triển nhiều game từ một game engine.

Hiện nay hầu hết các nhà phát triển game lớn và nổi tiếng như Vavle, Squa-Enix, Ubisoft, Gameloft…hoặc các nhóm phát triển game nhỏ thường xây dựng các engine cho riêng họ. Có một vài lý do mà một công ty game hay một nhóm phát triển game (studio) lựa chọn xây dựng thay vì mua công nghệ. Các studio này thường cần những yêu cầu cụ thể mà các phần mềm trung gian chưa đáp ứng được, việc tự xây dựng các chức năng này thường tốn ít thời gian hơn việc chờ các phần mềm trung gian đáp ứng các yêu cầu này. Ngoài ra các lập trình viên trong các nhóm phát triển game do không hiểu đầy đủ về công nghệ mà họ không xây dựng cũng phải rất cảnh giác trước nhiều lỗi không mong muốn. Thêm nữa các nhóm phát triển nhỏ có thể không đủ chi phí hoặc nghĩa vụ hợp đồng trả cho các phần mềm trung gian.

Hiện nay trên thị trường có nhiều engime phổ biến cho game mobile như Unity, Cocos2dx… nhưng để đào sâu tìm hiểu cốt lỏi và thao tác thành thạo một engine cần tốn khá nhiều thời gian và công sức. Cùng với việc bùng nỗ game trên nền tảng di động và bước đầu bước chân vào tìm hiểu thế giới game rộng lớn, em chọn xây dựng cho mình một game engine đa nền tảng và áp dụng xây dựng trò chơi ‘Kill The Rat’ mang tính giải trí đơn thuần.

* 1. **Mục Tiêu và Nhiệm vụ**

**Mục tiêu:** Xây đựng được một game engine đa nền tảng, từ đó áp dụng xây dựng một game 3D.

**Nhiệm vụ:**

* Tìm hiều lập trình OpenGL ES 2.0
* Tìm hiểu cấu trúc 1 game
* Thiết kế mô hình hệ thống game
* Xây dựng game engine
* Thiết kế game Kill The Rat mang tính giải trí đơn thuần
* Triển khai game sang nền tảng di động ( Android )
  1. **Đối tượng nghiên cứu và phạm vi hệ thống**

**Đối tượng:**

* Mô hình hệ thống game engine đa nền tảng
* Hệ điều hành window và tài nguyên hệ thống Android

**Phạm vi:** Đề tài chỉ dừng lại tại phần xây dựng phần vẽ (render) cho game engine và triển khai trò chơi “Kill The Rat” trên nền tảng PC (Window) và mobile (Android)

**1.5**

Chương 1 : Trình bày các khái niệm về Game Enigne, cách tiếp cận và phương pháp sử dụng để triên khai hệ thống Game Engine đa nền tảng. Chương này cũng trình bày mô hình chung của một Game Engine và một số thành phần cơ bản khác của hệ thống. Đồng thời trình bày một cách tổng quan nhất về các lý thuyết trong OpenGL, được áp dụng để xây dựng nên Render Engine

Chương 2 : Phân tích và thiết kế hệ thống Game Engine. Trình bày các mối quan hệ giữa các mô đun trong hệ thống, cũng như quan hệ giữa các thành phần trong cùng mô đun

Chương 3 : Ứng dụng Game Engine vào việc xây dựng trò chơi “Kill The Rat”

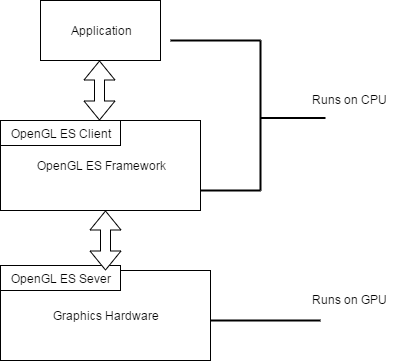
Chương 4 : Kết luận và hướng phát triển

**Chương 2: Cở Sở Lý Thuyết**

**2.1.** **Tổng quan về Game Engine**

* + 1. **Khái niệm Game Engine**

Game Engine là một công cụ hỗ trợ, một lớp trung gian ở giữa các ứng dụng game và nền tảng bên dưới, các thư viện lập trình cấp thấp. Game Enigine giúp phát triển ứng dụng game một cách nhanh chóng và đơn giản, đồng thời cung cấp khả năng tái sử dụng mã nguồn cao do có thể phát triển nhiều ứng dụng game khác nhau từ một game Engine.



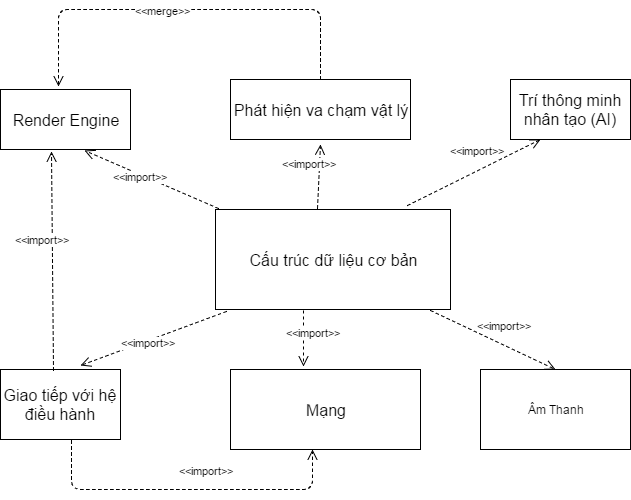
Hình 1: Kiến trúc tổng quan của Game Engine OpenGL ES

* + 1. **Mô hình Game Engine**

Game Engine hỗ trợ đa nền (Window PC và Mobile Android).

Đầu tiên là khả năng chạy đa nền. Để đạt được điều này, Engine sử dụng các bản build khác nhau trên các nền tảng khác nhau (chứ không phải sử dụng thông dịch). Để giải quyết vấn đề này, đầu tiên em chọn xây dựng một engine để phát triển game bằng C++ sử dụng công cụ Visual Studio 2013 trên nền tảng window. Sau đó sử dụng NDK trên android để chuyển engine này sang Android (NDK cho phép sử dụng C++ để phát triển ứng dụng) và triển khai game ra Mobilde Android.

Engine được thiết kế hướng đối tượng và yêu cầu hiệu năng chạy cao nên việc sử dụng ngôn ngữ C++ ngôn ngữ đáp ứng điều kiện trên.

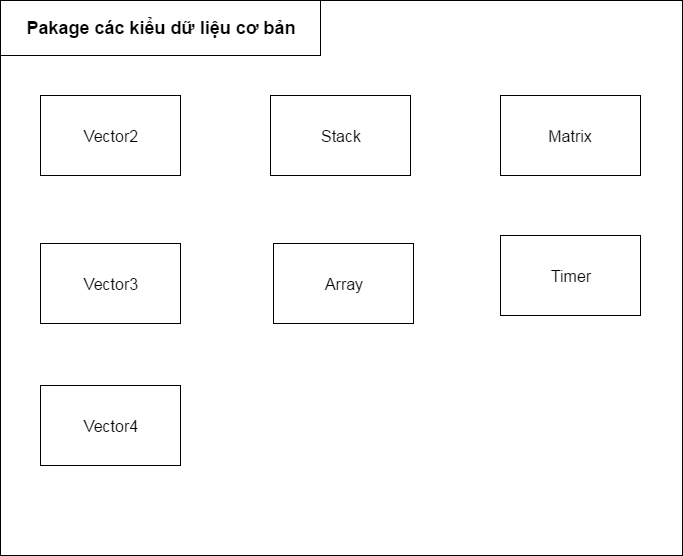


Kiến trúc tổng thể của Game Enigne

Do giới hạn về mặt thời gian, nên phạm vị đồ án này em tập trung vào các thành phần sau: Các kiểu dữ liệu cơ bản, Render Engine và âm thanh. Những thành phần này đủ để hỗ trợ người sử dụng tạo ra một game 3D và tương tác với chúng.

2.1.3 **cấu trúc dữ liệu cơ bản**

Các kiểu dữ liệu cơ bản bao gồm các cấu trúc dữ liệu cơ bản như mảng động, vector, ma trận…, và các phép toán trên các kiểu dữ liệu đó, cung cấp cho các thành phần khác sử dung.



Hinh . Biểu đồ lớp của thành phần dữ liệu cơ bản

Các kiểu dữ liệu cơ bản được chia làm các nhóm chính:

* Các yếu tố trong không gian 3D:
* Vector2, Vector3, Vector4: Lần lượt là các loại vector biểu diễn tọa độ 2 chiều, 3 chiều và tọa độ đồng nhất.
* Matrix: ma trận được sử dụng để biểu diễn các phép biến đổi: dịch, xoay, co giãn trong không gian 3 chiều.
* Kiểu đối tượng lưu trữ

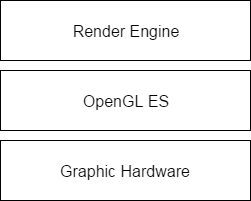
Array: Mảng động có thể tùy biến với hệ số mở rộng.

Stack : đặc tả ngăn xếp

* Xử lý thời gian trong game:
* Timer: gồm các hàm xử lý FPS (frame per second) và tính toán thời gian thực trong game
  + 1. **Thành phần Render Enginer**

Render Engine là thành phần cốt lõi của một Game Engine. Nó hỗ trợ người dùng các công việc thiết yếu để tạo ra một khung cảnh 3D. Người dùng sẽ không cần biết nhiều đến những công việc tầng thấp như quá trình đọc file tài nguyên, sủ dụng các API đồ họa 3D, tạo các hiệu ứng… mà quản lý logic các đối tượng bằng các lớp mà Render Engine cung cấp.

Đây là thành phần duy nhất trong Engine giao tiếp với phần cứng đồ họa thông qua thư viện đồ họa 3D cấp thấp, cụ thể ở đây là OpenGL ES.



Hinh Kiến trúc phân tầng của Render Engine

Thành phần này được thiết kế thành các mô đun chính như sau:

* Resource Manager là mô đun quản lý tài nguyên cần thiết để xây dựng nên các Object và cảnh 3D như texture 2D , cubemap… Mô đun này giúp cho việc dễ dàng quản lí tài nguyên , tăng tốc độ game và tránh việc sử dụng các model , texture trùng nhau
* Scene Manager là mô đun quản lý các màn chơi trong game (Scene), bao gồm các hàm chuyển màn chơi, pause ..
* Hiệu ứng ánh sáng và vật liệu giúp việc mô phỏng ánh sáng và màu sắc trong game.
* Mô phỏng chuyển động của động tượng là mô đun quản lý chuyển động của các vật thể ( animated )

Trong đồ án này em tập trung vào việc thực hiện 2 mô đun chính là Quản lý tài nguyên (Resource Manager) và quản lý cảnh (Scene Manager).

2.4 . Các thành phần còn lại

Phát hiện va chạm và tính toán vật lý là thành phần tính toán mô phỏng vật lí, phát hiện va chạm giữa các vật thể và phản hồi.

Trí thông mình nhân tạo là thành phần xử lí các công việc cần trí thông minh nhân tạo như tự động tìm đường, tạo ra các quái vật có thể tự điều khiển hành vi, …

Âm thanh là thành phần quản lý âm thanh trong game.

2.2.3 **Tổng quan về OpenGL**

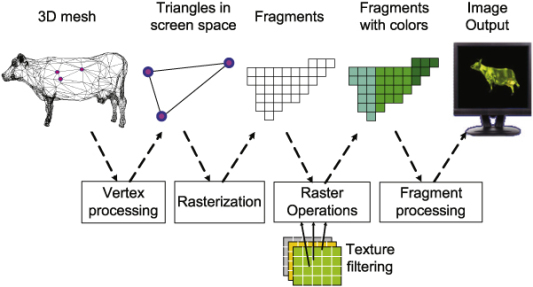
OpenGL (Open Graphics Library) là một tiêu chuẩn kỹ thuật đồ họa có mục đích định ra một giao diện lập trình ứng dụng (API) đa nền tảng (crossplatform) dùng để vẽ (render) đồ họa 3 chiều (3D). OpenGL cũng có thể được dùng trong các ứng dụng đồ họa 2 chiều (2D). Giao diện lập trình này chứa khoảng 250 hàm để vẽ các cảnh phức tạp từ những hàm đơn giản. Các API này tương tác trực tiếp với GPU (nhân xử lí đồ họa) để tăng tốc hiệu năng cho phần cứng.

Hiện nay OpenGL đã phát triển đến phiên bản 4.5, hỗ trợ cho các dòng GPU mới nhất.

Trong đồ án này em sử dụng OpenGL ES một giao diện lập trình ứng dụng (API) của OpenGL được sử dụng cho các hệ thống nhúng như điện thoại di động, thiết bị hỗ trợ cá nhân kĩ thuật số (PDA),...

**2.3.1** **Rendering pipeline**

Đây là 1 khái niệm căn bản trong 3D Graphics, cần hiểu rõ trước khi viết bất kì chương trình 3D nào. Renderring pipeline/Graphics pipeline là 1 quá trình gồm 1 chuỗi các bước để tạo ra ảnh bitmap(2D) của 1 scene 3D. Chúng ta biết rằng hình ảnh hiển thị trên màn hình thực chất là 1 ảnh 2 chiều gồm nhiều pixels, tương ứng với số pixels của màn hình (vd 1920x1080 pixels), màn hình có 1 tần số quét(vd 60hz) tương ứng với số lần refresh trong 1s, trong mỗi lần refresh thì ảnh bitmap trên màn hình được cập nhật và chúng ta có được hình ảnh liên tục không bị ngắt quãng.  
hình ảnh sau cho chúng ta khái niệm tổng quan về graphics pipeline của OpenGL :



Hình Khái quát về Rendering pipeline



Hình Biểu đồ các quy trình trong Rendering pipeline

* Vertex specification

Ở bước này, việc của chúng ta là cấp dữ liệu đầu vào cho OpenGL. Dữ liệu này có thể là dữ liệu liên quan tới đỉnh (vertex) của các hình cơ bản - tam giác, tứ giác, hoặc chỉ đơn giản là 2 đỉnh (vertices) của một đoạn thẳng. Chúng ta chỉ cần cung cấp tọa độ điểm, và (kèm với một số yếu tố khác), OpenGL sẽ lập ra các tam giác/tứ giác dựa trên những điểm này (việc này sẽ xảy ra ở bước Primitive assembly, xem ở dưới).

Ví dụ cấp tọa độ đỉnh đầu vào của một tam giác:

Const GLfloat vertices[]= {

1.0f,0.0f,0.0f,

0.0f,1.0f,0.0f,

0.0f,1.0f,0.0f

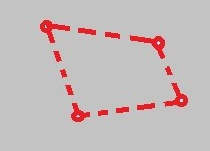
};

* Vertex shader

Trong giai đoạn vertex shader, dữ liệu đầu vào (dữ liệu về vị trí của các vertices), sẽ được tính toán để cho ra vị trí cuối cùng của vertices. Các vertices này chưa thể nằm được trên màn hình người sử dụng, ít nhất là cho đến giai đoạn "Rasterization" (sẽ nói sau).

Một ví dụ đơn giản như sau : chẳng hạn bạn đưa dữ liệu đầu vào là tọa độ của 1 cái hộp, không bị biến dạng. Nhưng bạn muốn nhìn cái hộp đó từ một góc 45 độ, nghĩa là, lúc đó tọa độ các vertices của cái hộp sẽ phải thay đổi, nghĩa là cái hộp sẽ phải bị biến dạng. Như vậy, dữ liệu đầu vào cần được "biến đổi" (transformed) sao cho khi xuất hiện trên màn hình, nó sẽ được đặt ở vị trí phù hợp với góc 45 độ mà bạn muốn nhìn. Như vậy, chúng ta sẽ nhận dữ liệu đầu vào là tọa độ gốc của cái hộp, biến đổi tọa độ đó thành tọa độ mới mà bạn muốn nhìn, và đó là công việc của vertex shader - tính toán vị trí cuối cùng của vertices

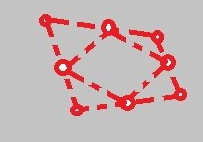
Vertex Shader chỉ xử lý dữ liệu về tọa độ vertices, những dữ liệu khác như colors, normals, textures..sẽ không mất đi mà chỉ đơn giản là bị bỏ qua, nghĩa là vertex shader sẽ truyền thẳng (pass-through) các dữ liệu này cho các stages sau mà không đá động gì tới chúng. Một ví dụ điển hình là fragment shader sẽ phụ trách dữ liệu liên quan tới color và texture của một vertex.



Hinh Minh họa kết quả sau quá trình vertex shader

* Tessellation

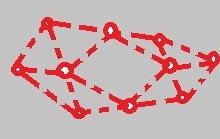
Tessellation là quá trình chia nhỏ các vertices. Thay vì phải tự tay sắp đặt các vertices nhỏ và rất nhỏ để tạo được độ chi tiết (detail, realistic, v..v..) cho model, OpenGL sẽ tự động làm việc này. Tessellation tiết kiệm được rất nhiều thời gian cho artists/level designers khi phải thiết kế nhân vật/màn chơi, cũng như giúp giảm tải GPU do không phải gọi API liên tục với số lượng lớn.



Hình Minh họa kết quả của quá trình Tessellation

* Geometry shader

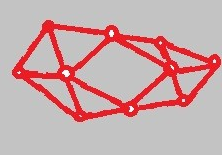
Trong giai đoạn này, các primitives mới sẽ được tạo ra từ primitives đầu vào. Khác với vertex shader stage chỉ truy cập duy nhất 1 vertex (per-vertex) trong 1 lần chạy shader, geometry shader có thể truy cập một geometric primitive (per-vertex hoặc multi-vertex) và do đó, có thể thực hiện xóa bỏ, tạo, chỉnh sửa các primitives này.



Hinh Minh họa kết quả quá trình Geomatry Shader

* Primitive assembly

Vì dữ liệu chúng ta truyền vào chỉ là tọa độ điểm của các vertices, nên trong stage này, OpenGL sẽ tìm cách nối các điểm đó lại để tạo thành các primitives cơ bản, có thể là điểm, đoạn thẳng, tam giác, hoặc tứ giác



Hinh Minh họa kết quả quá trình Primitive assembly

* Post- vertex processing

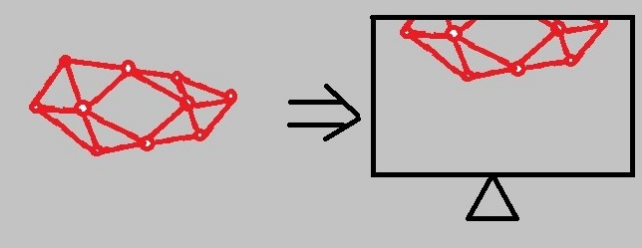
Đây là một nhóm các bước, nhưng chúng ta chỉ nói riêng clipping. Ở bước này, OpenGL sẽ cắt bớt những gì nằm bên ngoài khung nhìn (viewport).

Khung nhìn mặc định là hình chữ nhật áp sát màn hình của bạn, và bị giới hạn bởi cửa sổ nơi mà việc render được thực hiện.

Vị trí và kích thước khung nhìn được định nghĩa bởi 1 API call, "glViewport()", nhưng nếu không gọi, thì viewport mặc định là toàn bộ hình chữ nhật trong phạm vi cửa sổ chương trình.

Clipping được thực hiện tự động bởi OpenGL.

Tóm lại, ở bước này, những vertices nằm ngoài viewport sẽ bị xóa bỏ và các vertices liên quan tới vertices bị xóa sẽ được điều chỉnh và tạo thành primitive mới sao cho khớp, giống như đang bị cắt thật sự.

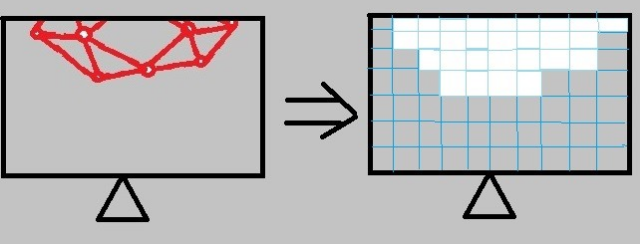


Hinh Minh họa kết quả quá trình Post- vertex processing

* Rasterization

Vậy là công việc với vertices đầu vào đã hoàn tất. Việc tiếp theo là biểu diễn vị trí các vertices này trên màn hình. Đó là công việc của rastersizer.

Rasterization lấy mỗi tam giác, cắt và loại bỏ các phần bên ngoài màn hình và chia các phần còn lại có thể nhìn thấy thành các mảng có kích thước pixel. (fragments). Rasterizer sẽ pha trộn những màu sắc trên bề mặt như thể hiện trong sơ đồ sau



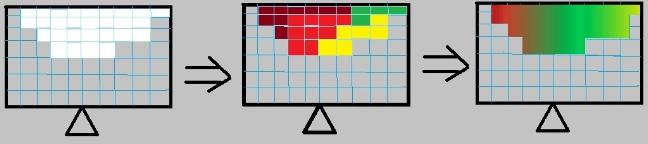
Hinh Minh họa kết quả quá trình Rasterization

* Fragment shader

Stage thứ hai và cũng là stage bắt buộc cuối cùng trong rendering pipeline.

Trong stage này, fragment shader sẽ lấy từng fragment từ rastersizer và tính toán các giá trị depth, stencil và màu của fragment đó. Trong stage này, màu của một fragment sẽ được quyết định, dù trong stage sau (8), màu có thể bị biến đổi một lần nữa.

Một điểm mạnh của fragment shader là chúng có thể được dùng với kĩ thuật texture mapping để áp texture lên bề mặt vật thể. Thay vì tô màu, chúng ta có thể "tô" texture.



Hinh Minh họa kết quả quá trình fragment shader

* 1. **Định nghĩa Vertex**

Một hình phẳng được xác định bởi một tập các vertex. Các vertex nối với nhau bằng các đoạn thẳng. Khi tất cả các vertex được nối thì hoàn thành hình dạng.

Để quản lý các kiểu hình dạng khác nhau trong một chương trình được dễ dàng, ta định nghĩa đơn giản cấu trúc cho kiểu dữ liệu vertex như sau

struct Vertex

{

GLfloat x,y,z ;

};

Cấu trúc này đơn giản lưu trữ các tọa độ đề các của vertex.

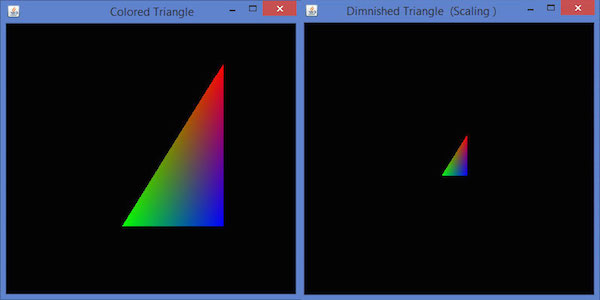
**Các phép biến hình (Transformations)**

Một hình phảng được tạo bằng cách định nghĩa các vertex trong tọa độ đề các, mapping giữa tọa độ đề các và tọa độ màn hình, rồi vẽ các đoạn thẳng nối các vertex. Để vẽ hình ở mọi nơi trên màn hình và theo mọi hướng, hình ảnh cần được thao tác theo nhiều cách khác nhau. Các thao tác như vậy gọi là các phép biến hình, bao gồm tịnh tiến, co giãn và quay.

Để di chuyển các đối tượng (Object) trong môi trường 3D chúng ta có thể di chuyển bằng cách thay đổi các Vertex của chúng và định lại bộ đệm trong mỗi khung hình (frame) nhưng điều này khá là phức tạp và chi phí cho việc này là khá cao. Có nhiều cách tốt hơn để biển đổi một đối tượng đó là sử dụng cách nhân các ma trận.

* + 1. **Co Dãn (Scaling)**

Co dãn (scale) là kĩ thuật được sử dụng để tăng hay giảm kich thước của một đối tượng



Hình Giảm kích thước tam giác đi 5 lần nhờ phép scale

Khi co dãn một vector, ta sẽ tăng độ dài vector và giữ nguyên hướng của chúng.

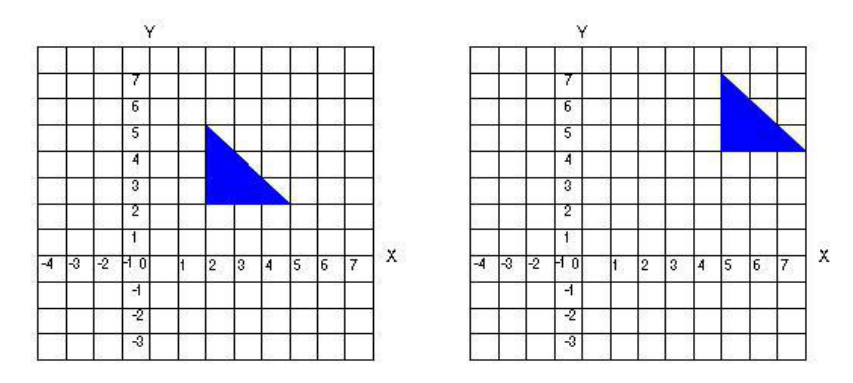
Trong môi trường 3D ứng với trường hợp 2D ở trên tương ứng với trục z là 1.

Để co dãn một vector (Px,Py,Pz,Pw) ta định nghĩa một vector S(Sx,Sy,Sz) trong đó Sx,Sy,Sz lần lượt là tỉ lệ scale theo trục X, Y, Z, sau đó định nghĩa một ma trận biển đổi tỉ lệ M (Scale Matrix) dựa trên vector S. Vector Scale P’ bằng tích của ma trận M và vector P:

P’ = M \* P

* + 1. **Tịnh tiến (Translation)**

Tịnh tiến một hình đến vị trí mới đơn giản là cộng hoặc trừ tọa độ của mỗi Vertex với một giá trị. Tam giác trong hình được tịnh tiến 3 đơn vị theo trục X và 2 đơn vị theo trục Y.



Hình. Tịnh tiến một tam giác

Tịnh tiến là quá trình di chuyển vector bằng viêc cộng thêm một vector vào vector gốc.

Trong không gian 3D để tịnh tiến một vector vector (Px,Py,Pz,Pw) ta định nghĩa một vector T(Tx,Ty,Tz) với Tx, Ty, Tz lần lượt là các đoạn cần dịch chuyển trên trục X, Y, Z. Kết quả sau khi tịnh tiến vector P’ bằng tích của ma trận tịnh tiến (Translate Matrix ) được tạo nên dựa vào vector T và vector P.

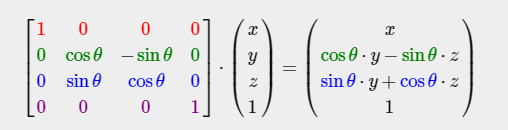
P’ = M \* P

**2.3.9 Xoay (Rotate)**

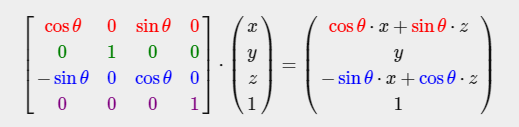
Xoay (Rotation) trong hệ trục tọa độ 3D được xác định bởi một góc quay và trục quay.

Ta sẽ sử dụng ma trận Rotate được tạo bằng cách sau để xoay quanh các trục tọa độ.

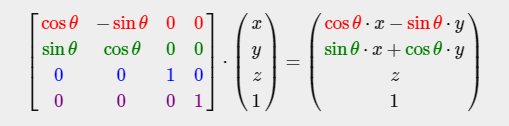
Xoay quanh trục X:



Xoay quanh trục Y:



Xoay quanh trục Z:



**2.3.10 Kết hợp các phép biến hình**

Nếu ta muốn kết hợp nhiều phép biến đổi trên cùng một đối tượng, ta phải nhân các ma trận biến đổi lại theo một thứ tự nhất định để không thay đổi hình dạng ban đầu của đối tượng.

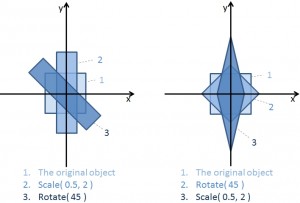
Các ma trận phải được nhân theo thứ tự như sau:

Scale – Rotation - Translation

trong đó Rotation = Rz \* Rx \* Ry.

Rx, Ry, Rz lần lượt là ma trận Rotate xoay quanh các trục x, y, z

Dưới đây là ví dụ về hình dạng của đối tượng theo các thứ tự nhân ma trận khác nhau:



Hình. Minh họa phép biến đổi theo 2 thứ tự

Tịnh tiến – Co dãn – Xoay và

Tịnh tiến – Xoay – Co dãn

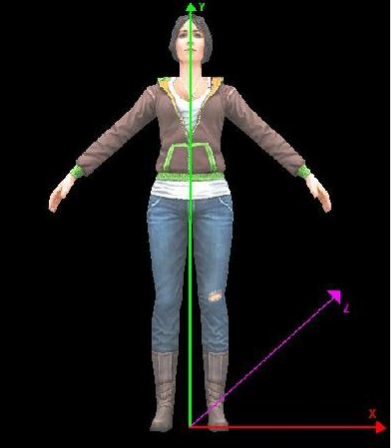
**2.3.11 Hệ thống các trục tọa độ (Coordinate System)**

Việc chuyển đổi các trục tọa độ tới tọa độ của thiết bị (NDC) và tọa độ màn hình thường được thực hiện theo từng bước. Trong đó ta chuyển tọa độ đỉnh của một đối tượng sang một vài hệ tọa độ trước khi chuyển chúng thành tọa độ màn hình. Ưu điểm của việc chuyển đổi sang một số hệ tọa độ trung gian là một số phép tính toán dễ dàng hơn trong các hệ tọa độ nhất định sẽ trở nên rõ ràng hơn.

Để chuyển đổi trục tọa độ từ một không gian (space) sang một trục toa độ không gian khác ta sử dụng một số các ma trận chuyển đổi, trong đó quan trọng nhất là ma trận mô hình (Model), hướng nhìn( View) và chiếu (Projection). Sau đây là một cái nhìn tổng quan về việc biến đổi thông qua các ma trận này.

Hinh . Biến đổi góc nhìn từ hệ tọa độ mô hình sang hệ tọa độ màn hình

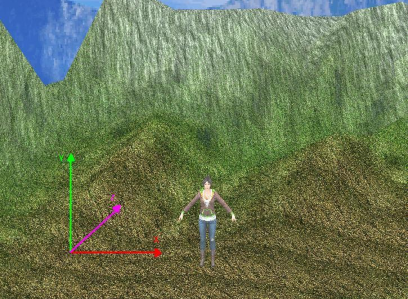
* Hệ tọa độ cục bộ (local space) là hệ tọa độ của đối tượng . Nơi mà các đối tượng (Object) được tạo ra.



Hinh Hệ trục tọa độ cục bộ (local space)

* Đối tượng từ hệ tọa độ cục bộ được chuyển sang hệ tọa độ thế giới ta bằng cách:

World Matrix \* Object



Hinh Đối tượng được đặt trong hệ tọa độ thế giới (World Space)

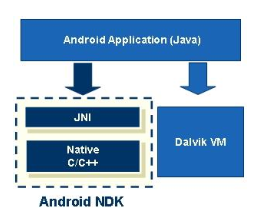
* Tiếp theo là quá trình biến đổi từ hệ tọa độ thế giới sang hệ tọa độ góc nhìn theo cách mà tọa độ này được nhìn thấy thông qua các camera hoặc từ điểm quan sát của người xem.
* Sau đó các tọa độ trong khung nhìn (view space) sẽ được co dãn sang trong pham vị -1.0 đến 1.0 để xác định xem đỉnh nào sẽ được xuất hiện trên màn hình
* Cuối cùng ta biến đổi thành tọa độ màn hình trong quá trình biển đổi khung nhìn. Biến đổi các tọa độ từ -1.0 và 1.0 sang phạm vi tọa độ được xác định bởi glViewPort. Kết quá sau đó sẽ được gửi đến các rasterizer để biến chúng thành các mảnh (fragment).

**2.4 Tổng quan về Android NDK**

**2.4.1 Android NDK là gì?**

Andoroid NDK là một bộ công cụ cho phép thực thì các thành phần của ứng dụng sử dụng ngôn ngữ thuần (Native) như C hoặc C++.

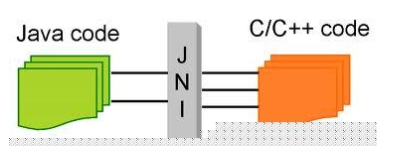
Android NDK giúp cho việc tái sử dụng mã nguồn C++, tăng hiệu năng ứng dụng , đồng thời giúp cho việc truy cấp vào hệ thống mà không phải thông qua máy ảo JVM



Hinh Các thành phần của Android NDK

**2.4.2. JNI là gì?**

JNI (Java Native Interface) là một framework cho phép gọi các hàm Java trong JVM từ các ngôn ngữ cấp thấp như C, C++ hay assembly. Nói nôm na dễ hiểu là khi ta muốn gọi Java từ C++ thì mình sẽ gọi thông qua JNI.



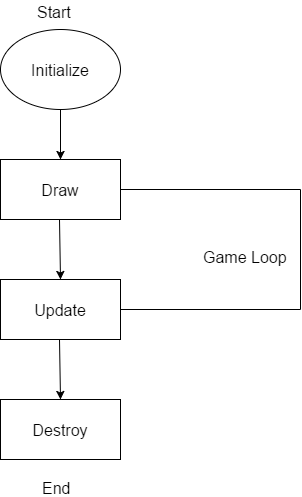
Hình Ứng dụng của JNI

**Chương 3 Phân tích thiết kế hệ thống**

**3.1 Kiến trúc tổng thể của hệ thống game**

Phần cốt lõi của hầu hết các trò chơi chính là vòng lặp được dùng để cập nhật và hiển thị trạng thái của trò chơi.

Minh họa:



Hình Lưu đồ vòng lặp game

Initialize (khởi tạo) thường dùng để tải tài nguyên, hình ảnh, phong chữ, màn hình nền, hay thiết lập số lượng quân đich, vv…

Draw dùng để vẽ (render) các thành phần trong mỗi màn chơi lên màn hình. Nó có thể vẽ nhiều đối tượng khác nhau, hay thiết lập vị trí cho mỗi đổi tượng trong các màn chơi vv…

Update được dùng để cập nhật lại màn chơi. (Chẳng hạn sử lỹ xự kiện khi người chơi nhấn một phím trên màn hình để nhân vật nhảy, vv..).

Hai phương thức Draw và Update được gọi và thực hiện cho đến khi trò chơi kết thúc.

Destroy được dùng để thu hồi các vùng nhớ, hình ảnh,… sau khi trò chơi kết thúc.

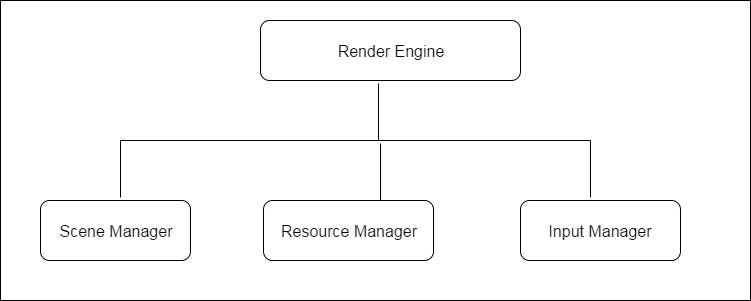
1. **Các thành phần chính của Render Engine**

Engine hỗ trợ người dùng các công việc thiết yếu để tao cảnh 3D và các mô hình 3D. Người dùng sẽ không cần biết nhiều đến những công việc tầng thấp như quá trình đọc file tài nguyên, sử dụng các API đồ họa 3D, tạo các hiệu ứng… mà quản lý logic các đối tượng bằng các giao diện do Engine cung cấp.

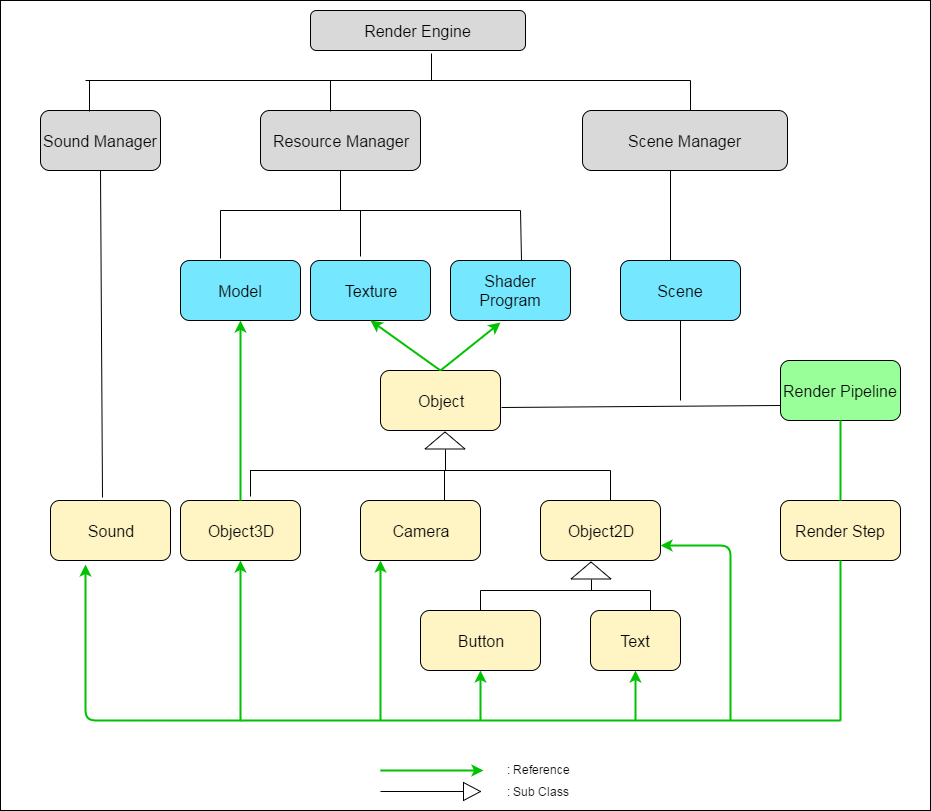
Engine giao tiếp với phần cứng đồ họa thông qua thư viện đồ họa 3D cấp thấp, cụ thể ở đây là OpenGL .

Render Engine được chia làm 3 mô đun chính là:

* Resource Manager : quản lý tài nguyên, các model ,object , texture trong game
* Scene Manager: quản lý các màn chơi (Scene)
* Input Handler : quản lý các tương tác với người dùng trong quá trình chơi game

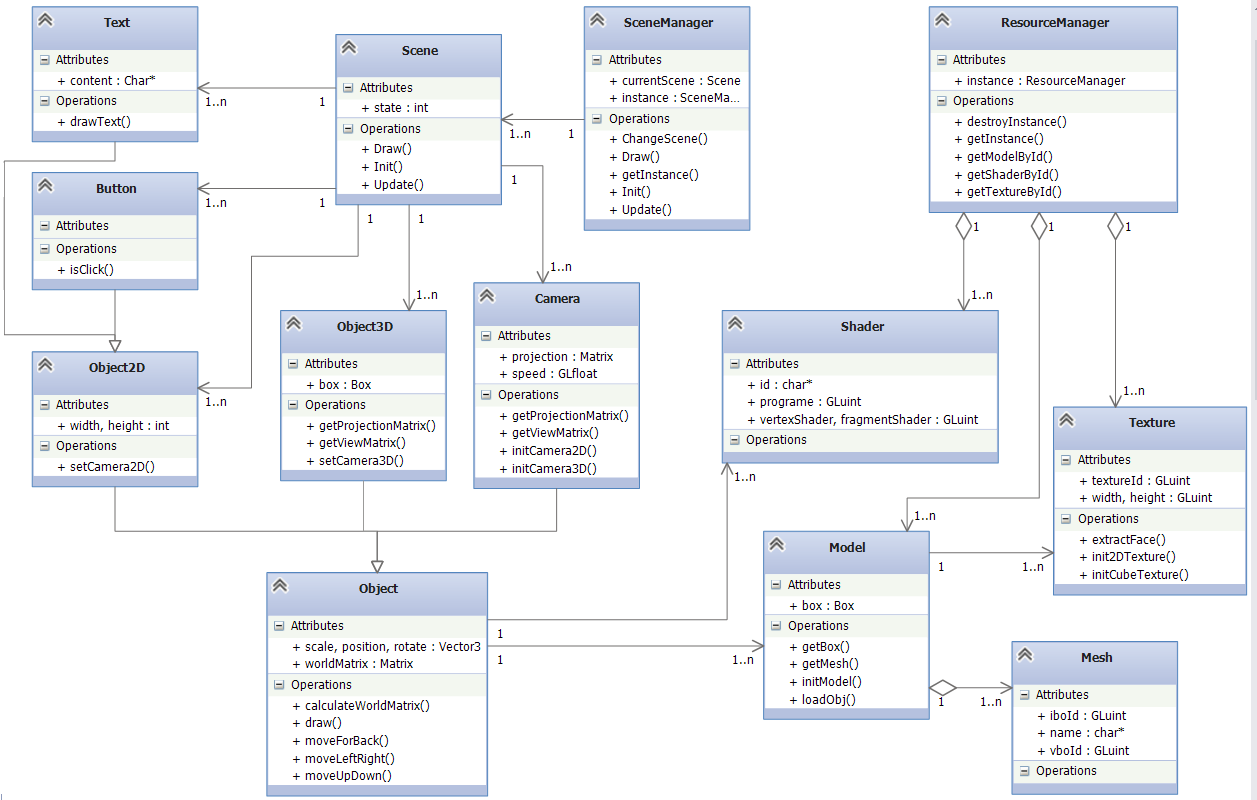


Hinh. Các mô đun chính trong Render Engine

****

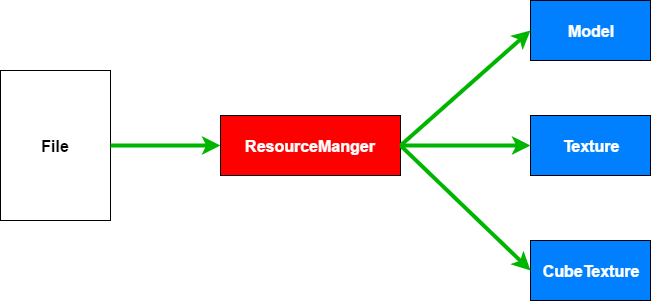
**Hình .Kiến trúc tổng quan của Render Engine**

**3.1 Sơ đồ lớp chi tiết của Game Engine**



Hình : OpenGL ES 2.0 Game Engine Class Diagram

* Resource Manager là mô đun quản lý tài nguyên gồm model, texture2D, Font,Shader. Các tài nguyên này được cung cấp cho việc xây dựng nên các mô hình 3D (Object3D) hay mô hình 2D (Object2D). Mô đun này sẽ được gọi đầu tiên trong hệ thống game. Lợi ích của việc quản lý tất cả tài nguyên như này tránh việc sử dụng trùng lặp bộ nhớ và tăng tốc độ game.
* Scene Manager là mô đun quản lý các màn chơi trong game (Scene), bao gồm các hàm khởi tạo màn chơi hay chuyển màn chơi (Change Scene) , tạm dừng (pause), vv...
* Scene là một cảnh game level, chẳng hạn như các cảnh trong 1 game thường có chia làm Start Scene, Play Scene , GameOver Scene. Việc chia làm các Scene và thông qua Scene Manager giúp cho việc quản lý game trở nên dễ dàng hơn, dễ xây dựng hơn trong quá trình phát triển game.
* Object chia làm 2 loại là Object 2D và Object3D. Việc mô phỏng chuyển động của các Object này sẽ thông qua một ma trận worldMatrix , dựa vào các vector postion, scale, rotate mà ta sẽ mô phỏng chuyển động cho các Object này.
* Camera trong engine giống như góc nhìn ngoài đời thật. Ở các vị trí khác nhau sẽ cho ra các kết quả khác nhau khi nhìn các Object. Camera cũng là một object trong engine, và bằng vào việc di chuyển camera trong môi trường 3D sẽ cho ra các kết quả khác nhau khi nhìn vào một cảnh trong game.
* Một Object sẽ bao gồm nhiều Model và Texture và mỗi Model sẽ được tạo ra từ nhiều Mesh.
* Shader sẽ giúp chúng ta load các file shader tạo hiệu ứng màu sắc cho object hay định vị trí cho một object. Bằng việc này chúng ta chỉ việc thao tác với các tệp shader (fragment Shader và vertex shader) để tạo hiệu ứng cho object một cách đơn giản hơn.
  + 1. Kiến trúc nạp tài nguyên



Hinh Kiến trúc nạp tài nguyên từ File

Trong file ResouceManager.txt quản lý tài nguyên cần có những thông tin sau:

\* Số lượng các mô hình (Model) và danh sách các mô hình bao gồm các thông tin sau:

- ID mô hình

- Đường dẫn đến file mô hình với định dạng (.obj, .nfg, ..)

\* Số lượng các chương trình Shader và danh sách các Shader với các thuộc tính:

- ID Shader

- Đường dẫn đến các file vertexShader và fragmentShader của chương trình Shader

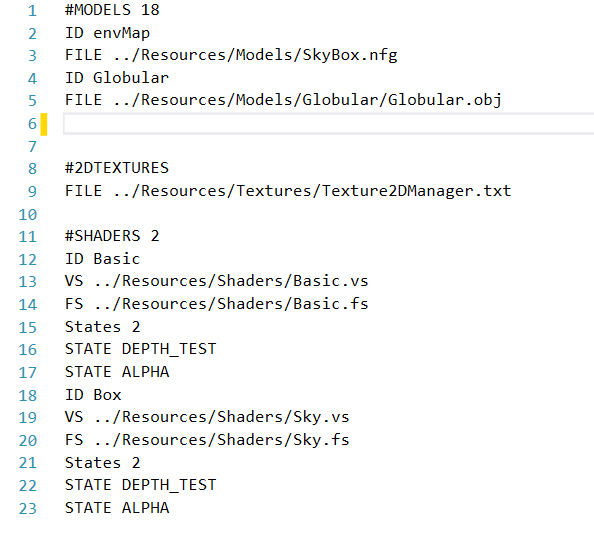
\* Đường đẫn đến file quản lý các Texture2D texture2DManager.txt. Trong file này sẽ bao gồm các đặc tả sau:

- Số lượng các Texture2D và danh sách các Texture2D với các thông tin:

+ ID và tên Texture2D

+ Tọa độ, chiều dài và chiều rộng của các texture2D

Ví dụ:



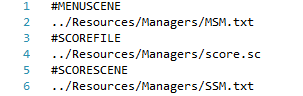
1.2.3. Kiến trúc nạp màn chơi



Hình. Kiến trúc nạp màn chơi từ file

Đầu tiên số lượng các màn chơi cũng như tên và vị trí lưu file đặc tả chi tiết cho mỗi màn chơi được lưu trong file SceneManager.txt.

Ví dụ:



Trong một file đặc tả màn chơi cần có những thông tin sau:

* Số lượng các Object3D và danh sách các Object3D cùng thuộc tính của chúng, bao gồm:

- Tên Object3D

- Số lượng mô hình (model) và danh sách các tên mô

- Số lượng Texture2D và danh sách tên các texture2D

- Số lượng CubeTextures và danh sách tên các cubeTextures

- Tên Shader

- Thiết lập vị trí, góc quay, độ co dãn ban đầu cho đối tượng qua các vector3 position, rotation, scale.

* Số lượng các Object2D và danh sách các Object2D cùng thuộc tính của chúng, bao gồm:

- Tên Object2D

- Số lượng Texture2D và danh sách tên các texture2D

- Tên Shader

- Thiết lập vị trí, góc quay, độ co dãn ban đầu cho đối tượng qua các vector3 position, rotation, scale.

* Số lượng các Button và danh sách các Button cùng thuộc tính của chúng, bao gồm:

- Tên Button

- Số lượng Texture2D và danh sách tên các texture2D

- Tên Shader

- Thiết lập vị trí, góc quay, độ co dãn ban đầu cho đối tượng qua các vector3 position, rotation, scale.

* Các thuộc tính của Camera3D bao gồm:
* Near
* Far
* Fov
* Tốc độ (speed)
* Các thuộc tính của Camera2D bao gồm:
* Near
* Far
* Tốc độ (speed)

Với tên các mô hình (model), texture và cubeTextures được tham chiếu từ file ResourceManager.txt**Chương 4. Ứng dụng phát triển Game Kill The Rat**

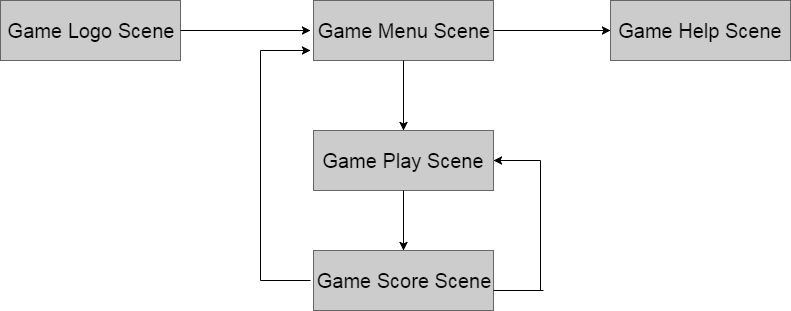
**4.1 Mô tả về game**

Kill The Rat là game 3D sử dụng game engine C++ OpenGL được thiết kế ở trên. Người chơi sẽ vào vai mèo Tom sử dụng số trứng được cho trong mỗi màn chơi tiêu diệt hết chuột trong phòng trước một khoản thời gian nhất định.

Trong mỗi màn chơi ngẫu nhiên người chơi sẽ nhận được quà, sử dụng các gói quà này để tăng thêm số lượng trứng hoặc thời gian trợ giúp cho người chơi vượt qua màn chơi.

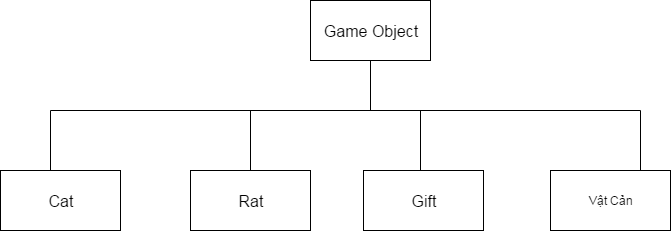
Trong mỗi màn chơi tốc độ của chuột (Rat) sẽ thay đổi tùy thuộc vào từng màn chơi. Mỗi màn chơi sẽ có các chướng ngại vật khác nhau để ngăn cản người chơi bắn trúng chuột.

**4.2 Cấu trúc các màn chơi trong game**



Hình Cấu trúc các màn chơi trong game

**4.3 Các đối tượng trong game**



Hinh Các đối tượng chính trong game

**4.4 Các quy luật chơi chính**

**4.4.1 Di chuyển**



Người chơi sử dụng 2 button mũi tên trái và phải để di chuyển mèo sang 2 bên trái, phải giúp cho việc bắn được dễ dàng hơn

**4.4.2. Xác định hướng bắn**

Khi người chơi chạm tay lên màn hình, game sẽ xác định và vẽ hướng bắn cho quả trứng bằng các chấm đỏ

Mèo Tom sẽ bắn trứng, trứng sẽ bay theo chuyển động ném xiên dưới tác dụng của trọng lực g = 10.

Tọa độ bay của quả trứng được xác định theo 3 trục toa độ như sau:

X = tọa độ x ban đầu + vân tốc trục x \* thời gian

Y = tọa độ y ban đầu + vận tốc ném trục y \* thời gian – 0.5 \* gia tốc trọng lực \* bình phương thời gian

Z = tọa độ z ban đầu + vận tốc trục z \* thời gian

**4.4.3. Các vật cản trong game**

Danh sách vật cản