**CHƯƠNG xxx. OpenGL ES**

1. **OpenGL ES là gì ?**

OpenGL ES (viết tắt cho OpenGL for Embedded Systems) là một giao diện lập trình ứng dụng (API) dùng để tạo ra cảnh 3D trên các thiết bị cầm tay và thiết bị nhúng. Hiện nay OpenGL ES là API đồ họa phổ biến nhất trên thiết bị di động, thậm chí nó còn có mặt trên desktop. Danh sách các nền tảng hỗ trợ OpenGL ES gồm iOS, Android, BlackBerry, bada, Linux và Windows. OpenGL ES cũng là cơ sở cho WebGL, một chuẩn cho đồ họa ba chiều trên web.

OpenGL ES 2.0 hỗ trợ iOS kể từ iPhone 3GS (ra mắt tháng 6/2009), hỗ trợ Android kể từ Android 2.0 (ra mắt tháng 3/2010). Phiên bản hiện tại là OpenGL ES 3.0. Hiện tại OpenGL ES 3.0 tương thích trên Android 4.3+ và trên iPhone 5s với iOS 7. OpenGL ES 3.0 có khả năng tương thích ngược với OpenGL ES 2.0, nghĩa là các ứng dụng được viết trên OpenGL ES 2.0 hoàn toàn có thể chạy được trên OpenGL ES 3.0.

OpenGL ES là một trong số các API được tạo ra bởi Khronos Group. Tổ chức này được thành lập năm 2000, chủ yếu tạo ra các API chuẩn và hoàn toàn miễn phí. Khronos Group cũng kiểm soát OpenGL, đây là một API đồ họa 3D chuẩn đa nền tảng dành cho desktop, tương thích với cả Linux, Mac OS X và Microsoft Windows.

Bởi vì OpenGL là một thư viện đồ họa 3D được sử dụng rộng rãi nên người ta mới dựa trên đó để phát triển một thư viện cho thiết bị di động. Do đó có thể coi OpenGL ES là một subset của OpenGL. Sở dĩ người ta phải tạo ra OpenGL ES thay vì dùng OpenGL vì những khác biệt trên thiết bị di động và desktop cũng như những hạn chế phần cứng trên thiết bị di động. Khi thiết kế OpenGL ES những phiên bản cũ (1.0, 1.1 và 2.0), người ta phải xem xét các hạn chế của thiết bị như khả năng xử lí, bộ nhớ, truyền dữ liệu, pin, … Đặc tả OpenGL ES dựa vào những tiêu chuẩn sau:

* OpenGL API rất đồ sộ và phức tạp, do vậy mục tiêu của nhóm thiết kế OpenGL ES là tạo nên một API thích hợp với thiết bị giới hạn phần cứng. Để làm được điều này, nhóm thiết kế đã loại bỏ những tính năng dư thừa của OpenGL. Một công việc mà có thể làm bằng nhiều cách khác nhau thì chỉ giữ lại cách hữu ích nhất và loại bỏ những kĩ thuật dư thừa. Ví dụ như khi đặc tả khối hình học, OpenGL có các tính năng như chế độ hiển thị tức khắc (immediate mode), hiển thị theo danh sách (display lists) hay lưu trữ đỉnh trong RAM (vertex arrays). Trong OpenGL ES, chỉ vertex arrays được giữ lại, còn immediate mode và display lists bị loại bỏ.
* Ngoài việc loại bỏ tính năng dư thừa trong OpenGL, OpenGL ES còn phải có khả năng tương thích với OpenGL. OpenGL ES được thiết kế sao cho các ứng dụng được viết và dùng một phần các tính năng của OpenGL cũng có khả năng chạy trong OpenGL ES. Điều này rất quan trọng bởi vì nó cho phép các lập trình viên tái sử dụng code đã viết.
* Các tính năng mới được giới thiệu để giải quyết một hạn chế nào đó trên thiết bị. Ví dụ như để giảm thiểu tốn pin và tăng tốc ứng dụng, ta có thể làm giảm độ chính xác của số thực (dùng precision qualifiers) trong shader.
* Các tính năng nâng cao chất lượng hình ảnh trong OpenGL ES rất ít. Vì vậy trước đây, để đảm bảo chất lượng các pixel được vẽ trên màn hình, các thiết bị di động bị giới hạn kích thước màn hình.
* Nhóm thiết kế OpenGL ES muốn đảm bảo bất cứ bản cài đặt OpenGL ES nào cũng phải đảm bảo tiêu chuẩn về chất lượng hình ảnh, sự chính xác và không có lỗi. Để đạt được điều này, nhóm đã tạo ra bài kiểm tra thống nhất mà bất cứ bản cài đặt OpenGL ES nào cũng phải vượt qua.

Khronos đã cho ra bốn phiên bản OpenGL ES: OpenGL ES 1.0, OpenGL ES 1.1, OpenGL ES 2.0 và OpenGL ES 3.0. OpenGL ES 1.0 và 1.1 dùng một tập lệnh cố định được cài sẵn trong GPU (fixed function pipeline) và tương ứng với OpenGL 1.3 và 1.5.

OpenGL ES 2.0 sử dụng shader (một ngôn ngữ lập trình cấp cao tương tự như C) để điều khiển quá trình render (programmable graphics pipeline) và tương ứng với OpenGL 2.0.

OpenGL ES 3.0 là một bước tiến hóa nữa của OpenGL ES trên thiết bị di động và dựa trên OpenGL 3.3. OpenGL ES 2.0 đã mang các tính năng giống với DirectX 9 và Microsoft Xbox 360 lên thiết bị di động, trong khi đó đồ họa tiếp tục phát triển mạnh mẽ trên desktop GPUs. Mặc dù vậy OpenGL ES 2.0 vẫn còn thiếu nhiều tính năng đồ họa cao cấp như đổ bóng (shadow mapping), render trong phòng (volume rendering), hiệu ứng particle trên GPU, render nhiều mô hình giống nhau (geometry instancing), nén ảnh (texture compression) và làm hình ảnh trung thực (gamma correction). OpenGL ES 3.0 đã mang các tính năng này lên thiết bị di động nhưng vẫn phải đảm bảo thỏa mãn các giới hạn phần cứng của thiết bị di động.

Dĩ nhiên, những giới hạn phần cứng trước đây không còn quá quan trọng khi thiết kế các phiên bản OpenGL ES hiện nay. Chẳng hạn thiết bị di động hiện nay đã có màn hình lớn (nhiều thiết bị còn có độ phân giải lớn hơn màn hình desktop). Thêm vào đó các thiết bị này sở hữu CPU đa nhân tốc độ cao và bộ nhớ ngày càng lớn. Do vậy khi phát triển OpenGL ES 3.0, Khronos chú trọng vào tính ứng dụng hơn là giới hạn phần cứng của thiết bị.

1. **Nguyên lí hoạt động của OpenGL ES**

**Hình xxx-2-1** Luồng hoạt động (pipeline) của OpenGL ES 2.0

Giải thích sơ lược về các giai đoạn:

1. **Vertex shader**

Vertex shader là một đoạn code được thực thi trên mỗi đỉnh của khối hình học. Đầu vào của vertex shader bao gồm:

* Attributes – Dữ liệu cho từng đỉnh (ví dụ tọa độ, màu sắc, normal vector, …)
* Uniforms – Là hằng số giống nhau cho tấc cả các đỉnh
* Samplers – Là một biến uniform dùng để đọc texture
* Shader program – Là mã nguồn vertex shader đã được biên dịch

Đầu ra của vertex shader được gọi là biến varying. Tại giai đoạn rasterization (phần c), giá trị varying được tính toán cho mỗi fragment (hay còn gọi là pixel) và trở thành input của fragment shader. Kỹ thuật để tính giá trị varying cho fragment dựa trên giá trị varying của đỉnh được gọi là nội suy (interpolation).

**Hình xxx-2-2** Vertex shader cho OpenGL ES 2.0

Vertex shader thường được dùng để thay đổi vị trí của một đỉnh (dùng ma trận), tính toán độ sáng tại đỉnh đó, tạo ra hoặc thay đổi tọa độ texture, …

Một ví dụ vertex shader:

|  |
| --- |
| // uniforms used by the vertex shader  uniform mat4 u\_mvpMatrix; // matrix to convert P from model  // space to normalized device space.  // attributes input to the vertex shader  attribute vec4 a\_position; // position value  attribute vec4 a\_color; // input vertex color  // varying variables – input to the fragment shader  varying vec4 v\_color; // output vertex color  void main()  {  v\_color = a\_color;  gl\_Position = u\_mvpMatrix \* a\_position;  } |

Shader này nhận vào vị trí và màu tại một đỉnh, thay đổi vị trí đỉnh đó bằng cách nhân với ma trận 4x4 và xuất ra vị trí mới và màu.

1. **Primitive assembly**

Sau giai đoạn vertex shader là giai đoạn primitive assembly. Một primitive là đối tượng hình học (chẳng hạn đoạn thẳng, tam giác thậm chí chỉ là một điểm) có thể được vẽ ra bằng cách sử dụng lệnh vẽ thích hợp trong OpenGL ES. Mỗi lệnh vẽ đặc tả một tập các thuộc tính của đỉnh và kiểu đối tượng hình học sẽ vẽ. Những thuộc tính của đỉnh chứa đựng thông tin để vertex shader tính toán vị trí và các thông tin khác sau đó truyền đến fragment shader. Những thông tin này có thể là vị trí, màu sắc hay tọa độ texture.

Tại giai đoạn primitive assembly, các đỉnh sẽ được hợp lại thành các đối tượng hình học như tam giác, đoạn thẳng hay point-sprite. Với mỗi đối tượng hình học, cần phải xác định xem nó có nằm trong vùng nhìn thấy hay không. Nếu đối tượng hình học chỉ nằm một phần trong vùng nhìn thấy thì nó sẽ bị cắt phần nằm bên ngoài (clipping). Nếu đối tượng hình học hoàn toàn nằm bên ngoài thì nó sẽ bị loại bỏ. Sau bước cắt bỏ, tọa độ 3D của đỉnh sẽ được chuyển thành tọa độ 2D trên màn hình. Một bước loại bỏ khác gọi là culling sẽ loại bỏ đối tượng hình học dựa vào việc đối tượng đó có xoay mặt vào hướng nhìn hay không. Sau khi clipping và culling, đối tượng hình học sẽ được đưa qua bước tiếp theo – rasterization.

1. **Rasterization**

Tại giai đoạn rasterization, các đối tượng hình học (point-sprite, đoạn thẳng, tam giác) sẽ được vẽ ra. Tiến trình này sẽ biến đổi các đối tượng hình học 3D thành tập các fragment 2D. Các fragment này sau đó sẽ được xử lí tiếp với fragment shader. Các fragment 2D này chính là các pixel sẽ được vẽ lên màn hình.

**Hình xxx-2-3** Giai đoạn rasterization của OpenGL ES 2.0

1. **Fragment shader**

Fragment shader là một đoạn code được thực thi trên mỗi fragment được tạo ra từ giai đoạn rasterization. Đầu vào của fragment shader bao gồm:

* Biến varying – Là đầu ra của vertex shader, sau đó được nội suy ở giai đoạn rasterization
* Uniforms – Là hằng số giống nhau cho tấc cả các fragment
* Samplers – Là một biến uniform dùng để đọc texture
* Shader program – Là mã nguồn fragment shader đã được biên dịch

Nhiệm vụ của fragment shader là loại bỏ hoặc gán màu sắc cho fragment thông qua biến gl\_FragColor.

**Hình xxx-2-4** Fragment shader cho OpenGL ES 2.0

Một ví dụ fragment shader:

|  |
| --- |
| precision mediump float;  varying vec4 v\_color; // input vertex color from vertex shader  void main(void)  {  gl\_FragColor = v\_color;  } |

1. **Per-fragment operations**

Sau fragment shader, giai đoạn tiếp theo là per-fragment operations. Một fragment được tạo ra từ giai đoạn rasterization với tọa độ màn hình (xw, yw) chỉ có thể chỉnh sửa pixel tại vị trí (xw, yw) trong framebuffer.

**Hình xxx-2-5** Per-fragment operations cho OpenGL ES 2.0

Per-fragment operations thực hiện các công việc sau cho mỗi fragment:

* Pixel ownership test – Việc kiểm tra này xác định xem một pixel tại vị trí (xw, yw) trong framebuffer có thuộc quyền sở hữu của ứng dụng OpenGL ES hiện tại hay không. Ví dụ như cửa sổ ứng dụng OpenGL ES hiện tại bị che một phần bởi cửa sổ ứng dụng khác, khi đó hệ thống sẽ xác định các pixel bị che khuất sẽ không thuộc quyền sở hữu của ứng dụng OpenGL ES và vì vậy các pixel này sẽ không được hiển thị.
* Scissor test – Việc kiểm tra này xác định xem một fragment có nằm trong một hình chữ nhật nào đó (hình chữ nhật này có thể được đặc tả trong ứng dụng) hay không. Nếu fragment nằm ngoài hình chữ nhật, nó sẽ bị loại bỏ.
* Stencil and depth tests – Kiểm tra giá trị stencil và depth của fragment để xác định xem có loại bỏ fragment này hay không.
* Blending – Bước này sẽ hòa trộn màu sắc của fragment đang xét với màu sắc đã có tại vị trí (xw, yw) trong framebuffer.
* Dithering – Bước này sẽ làm giảm đi những hiệu ứng hình ảnh không mong muốn do giá trị màu sắc được lưu trong framebuffer có độ chính xác thấp.

Sau giai đoạn này, một fragment sẽ bị loại bỏ hoặc giá trị màu sắc, depth, stencil của fragment sẽ được ghi vào framebuffer tại vị trí (xw, yw). Việc giá trị màu sắc, depth, stencil của fragment được ghi hay không phụ thuộc vào các bit mặt nạ tương ứng có được bật hay không. Hơn thế nữa, các bit mặt nạ còn có thể điều khiển cách ghi. Ví dụ như bit mặt nạ cho buffer màu sắc loại bỏ thành phần màu đỏ trước khi ghi vào buffer màu sắc.

OpenGL ES 2.0 còn cung cấp một phương thức cho phép đọc được các pixel từ framebuffer. Nhưng chỉ đọc được giá trị màu sắc, còn giá trị stencil và depth thì không.

1. **Cơ bản về lập trình với OpenGL ES 2.0**
2. **EGL**

Một ứng dụng OpenGL ES thì cần phải biết ngữ cảnh vẽ (rendering context) và bề mặt vẽ (drawing surface). Ngữ cảnh vẽ lưu trữ các trạng thái của OpenGL ES còn bề mặt vẽ là bề mặt để vẽ các đối tượng hình học lên. Bề mặt vẽ đặc tả kiểu của buffer được sử dụng trong quá trình vẽ như color buffer, depth buffer, stencil buffer.

OpenGL ES không quản lí việc tạo ngữ cảnh vẽ cũng như việc gắn ngữ cảnh vẽ với hệ thống mà ứng dụng đang chạy. EGL chính là cầu nối giữa API vẽ như OpenGL ES với hệ thống.

Trước khi vẽ bất cứ thứ gì ra màn hình, một ứng dụng OpenGL ES cần phải thực hiện các bước sau thông qua EGL:

* Khởi tạo hệ thống hiển thị trên thiết bị. Ví dụ điện thoại gập (flip phone) có thể có hai màn hình LCD và chúng ta có thể vẽ lên một hoặc cả hai màn hình.
* Tạo bề mặt vẽ. Bề mặt vẽ được tạo bởi EGL chia thành hai loại, đó là bề mặt hiển thị (on-screen surface) và bề mặt không hiển thị (off-screen surface). Bề mặt hiển thị được gắn với cửa sổ ứng dụng. Bề mặt không hiển thị là một buffer các pixel không hiển thị cho người dùng mà thường được ghi vào một texture.
* Tạo ngữ cảnh vẽ. Sau khi được tạo, ngữ cảnh vẽ cần phải được gắn với một bề mặt vẽ thích hợp.

Ngoài các tính năng được mô tả bên trên, EGL còn có các tính năng khác như quản lí pin, hỗ trợ nhiều ngữ cảnh vẽ trong một tiến trình, chia sẻ đối tượng (textures, vertex buffers) qua nhiều ngữ cảnh vẽ trong một tiến trình, lấy con trỏ hàm của các hàm EGL hay các hàm OpenGL ES mở rộng, …

Phiên bản mới nhất của EGL hiện nay là EGL 1.4.

1. **Thư viện và các tập tin header**

Để biên dịch được một ứng dụng OpenGL ES 2.0, ta cần phải liên kết với các thư viện sau: libGLESv2.lib cho OpenGL ES và libEGL.lib cho EGL.

Ngoài ra ta còn cần phải include các tập tin header sau trong code để sử dụng được OpenGL ES và EGL:

|  |
| --- |
| #include <EGL/egl.h>  #include <GLES2/gl2.h>  #include <GLES2/gl2ext.h> |

egl.h là tập tin header cho EGL, gl2.h là tập tin header cho OpenGL ES và gl2ext.h là các hàm OpenGL ES mở rộng chuẩn của Khronos.

Các tập tin header và thư viện có thể khác nhau trên mỗi nền tảng. Nhóm phát triển OpenGL ES cố gắng đưa ra một chuẩn chung cho cách tổ chức và đặt tên thư viện, tuy nhiên điều này không đúng cho tấc cả các nền tảng. Do vậy các nhà phát triển ứng dụng OpenGL ES nên tham khảo cách sử dụng thư viện và tập tin header cho mỗi nền tảng.

1. **Cú pháp câu lệnh của EGL**

Tấc cả tên hàm của EGL đều bắt đầu bằng tiền tố egl và các chữ cái đầu mỗi từ đều in hoa (ví dụ như eglCreateWindowSurface). Tương tự, tên các kiểu dữ liệu cũng bắt đầu bằng tiền tố EGL và cũng dùng chữ cái in hoa đầu mỗi từ (ngoại trừ EGLint và EGLenum).

|  |  |  |
| --- | --- | --- |
| **Kiểu dữ liệu** | **Kiểu dữ liệu trong C** | **Kiểu dữ liệu trong EGL** |
| Số nguyên 32-bit | int | EGLint |
| Số nguyên không dấu 32-bit | unsigned int | EGLBoolean, EGLenum |
| Con trỏ 32-bit | void \* | EGLConfig, EGLContext, EGLDisplay, EGLSurface, EGLClientBuffer |

**Bảng xxx-2-1** Các kiểu dữ liệu trong EGL

1. **Cú pháp câu lệnh của OpenGL ES**

Tấc cả tên hàm của OpenGL ES đều bắt đầu bằng tiền tố gl và các chữ cái đầu mỗi từ đều in hoa (ví dụ như glBlendEquation). Tương tự, tên các kiểu dữ liệu cũng bắt đầu bằng tiền tố GL.

Hơn nữa, việc đặt tên hàm còn xuất phát từ tính chất các tham số của hàm. Những tính chất này có thể là số lượng tham số (từ 1 đến 4) hoặc kiểu dữ liệu của tham số (byte [b], unsigned byte [ub], short [s], unsigned short [us], int [i], fixed [x] và float [f]) hoặc tham số có phải là kiểu vector (v) hay không. Hai ví dụ dưới đây cho thấy điều này:

Ví dụ 1: Hai hàm này tương đương, ngoại trừ một hàm cho biết kiểu của biến uniform là float, hàm còn lại là int.

|  |
| --- |
| glUniform2f(location, 1.0f, 0.0f);  glUniform2i(location, 1, 0); |

Ví dụ 2: Hai hàm này cũng tương đương nhưng một hàm truyền tham số vector, hàm còn lại thì không.

|  |
| --- |
| GLfloat coord[4] = { 1.0f, 0.75f, 0.25f, 0.0f };  glUniform4fv(location, coord);  glUniform4f(location, coord[0], coord[1], coord[2], coord[3]); |

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu** | **Kiểu** | **Kiểu trong C** | **Kiểu trong OpenGL ES** |
| b | Số nguyên có dấu 8-bit | signed char | GLbyte |
| ub | Số nguyên không dấu 8-bit | unsigned char | GLubyte, GLboolean |
| s | Số nguyên có dấu 16-bit | short | GLshort |
| us | Số nguyên không dấu 16-bit | unsigned short | GLushort |
| i | Số nguyên có dấu 32-bit | int | GLint |
| ui | Số nguyên không dấu 32-bit | unsigned int | GLuint, GLbitfield, GLenum |
| x | Số thực dấu chấm tĩnh 16.16 | int | GLfixed |
| f | Số thực dấu chấm động 32-bit | float | GLfloat, GLclampf |

**Bảng xxx-2-2** Các hậu tố trong tên hàm và kiểu tham số của hàm trong OpenGL ES

Ngoài ra OpenGL ES còn có kiểu GLvoid. Kiểu này cho phép truyền con trỏ vào trong hàm.

1. **Xử lí lỗi**

Các lỗi khi thực hiện một hàm trong OpenGL ES sẽ phát sinh một mã lỗi. Mã lỗi này được hệ thống ghi nhận lại và có thể lấy ra thông qua hàm glGetError. Các lỗi tiếp theo sẽ không được ghi nhận lại nếu lỗi hiện tại không được lấy ra thông qua hàm glGetError. Một khi mã lỗi được lấy ra, mã lỗi hiện tại sẽ được chuyển thành GL\_NO\_ERROR. Hàm gây ra lỗi sẽ bị chương trình bỏ qua và không gây crash ngoại trừ mã lỗi GL\_OUT\_OF\_MEMORY.

Bảng sau đây mô tả một vài mã lỗi thường gặp:

|  |  |
| --- | --- |
| **Mã lỗi** | **Mô tả** |
| GL\_NO\_ERROR | Không có lỗi kể từ lần cuối gọi hàm glGetError |
| GL\_INVALID\_ENUM | Tham số truyền vào nằm ngoài miền giá trị của enum. Hàm gây ra lỗi này sẽ bị bỏ qua. |
| GL\_INVALID\_VALUE | Tham số truyền vào nằm ngoài miền giá trị của kiểu dữ liệu. Hàm gây ra lỗi này sẽ bị bỏ qua. |
| GL\_INVALID\_OPERATION | Hàm này không thể thực hiện ở thời điểm hiện tại. Hàm gây ra lỗi này sẽ bị bỏ qua. |
| GL\_OUT\_OF\_MEMORY | Không đủ bộ nhớ để thực hiện hàm này. |

**Bảng xxx-2-3** Các mã lỗi thường gặp

1. **Lệnh Flush và Finish**

OpenGL ES kế thừa mô hình client-server từ OpenGL. Ứng dụng (hay client) sẽ đẩy các lệnh lên server và server sẽ thực thi những lệnh này. Trong OpenGL, client và server có thể nằm ở hai máy khác nhau và được kết nối thông qua mạng. OpenGL ES cũng cho client và server nằm ở hai máy khác nhau, nhưng bởi vì OpenGL ES hướng đến thiết bị cầm tay và thiết bị nhúng nên client và server thường chỉ nằm trên một máy.

Trong mô hình client-server, client không cần phải gửi từng lệnh lên server ngay lập tức. Nếu client và server được kết nối thông qua mạng, việc gửi từng lệnh là cực kì không hiệu quả. Thay vào đó, các lệnh sẽ được lưu tạm trong bộ nhớ đệm và sau đó sẽ gửi nhiều lệnh cùng lúc lên server. Điều này yêu cầu cần phải có một cơ chế cho phép client biết được server đã thực hiện xong gói lệnh trước đó hay chưa. Lệnh glFlush được dùng để đẩy tấc cả các lệnh đang chờ lên server. Chú ý là lệnh glFlush chỉ đẩy các lệnh lên server và không chờ các lệnh đó thực hiện xong. Nếu client yêu cầu các lệnh đó phải thực hiện xong thì ta sử dụng lệnh glFinish. Tuy nhiên chỉ sử dụng lệnh glFinish khi thật cần thiết. Bởi vì lệnh glFinish sẽ không trả về mãi cho đến khi tấc cả các lệnh được thực hiện xong. Điều này sẽ làm chậm đáng kể tốc độ của ứng dụng.

|  |
| --- |
| **void** glFlush(**void**)  **void** glFinish(**void**) |

1. **Quản lí trạng thái**

Ta đã biết một ứng dụng OpenGL ES khi thực thi là một luồng các giai đoạn (pipeline stages). Mỗi giai đoạn sẽ có những trạng thái có thể bật/tắt được và những trạng thái có giá trị thích hợp. Ví dụ như ta có các trạng thái: bật chế độ hòa trộn (blending enable), thành phần hòa trộn (blend factors), bật khử mặt khuất (cull enable) và mặt để khử (cull face). Những trạng thái này được khởi tạo với giá trị mặc định khi ngữ cảnh vẽ (EGLcontext) được khởi tạo. Việc bật/tắt các trạng thái thông qua hàm glEnable và glDisable. Ngoài ra chúng ta cũng có thể kiểm tra một trạng thái đã được bật hay chưa thông qua hàm glIsEnabled.

1. Abc

**CHƯƠNG XXX. Adreno SDK**

1. **Adreno SDK là gì?**

Adreno SDK là một bộ thư viện C++/miễn phí/mã nguồn mở hỗ trợ viết ứng dụng OpenGL ES đa nền tảng, nó hỗ trợ Android, Windows Phone, Symbian, WinRT, Windows, Linux, ... Đối với các nền tảng Desktop (như Windows, Linux), ta sẽ phải sử dụng bộ giả lập của Adreno SDK bởi vì các nền tảng này không hỗ trợ OpenGL ES. Thực chất Adreno SDK là một gói bọc (wrapper) các hàm của OpenGL cùng với đó là các công cụ hỗ trợ người phát triển ứng dụng OpenGL ES.

Adreno SDK do Qualcomm (nhà sản xuất chip đồ họa Adreno danh tiếng) phát triển. Phiên bản hiện tại là Adreno SDK v3.9.

**Link**: <https://developer.qualcomm.com/software/adreno-gpu-sdk/tools>.

1. **Cài đặt Adreno SDK**

Tải bộ cài đặt từ đường link bên trên về. Tiến hành cài đặt như thông thường. Sau khi cài đặt xong, ta sẽ thấy các thư mục sau:

* **Bin**: Bao gồm những file thực thi của tutorials, samples, demos và tools.
* **Development**: Bao gồm các thành phần sau:
  + **Tutorials**: Là những chương trình minh họa cách sử dụng một tính năng nào đó của OpenGL ES. Những chương trình này đơn giản và dễ hiểu hơn những sample. Và thích hợp cho những người mới bắt đầu tìm hiểu OpenGL ES.
  + **Samples**: Là những chương trình phức tạp hơn tutorial. Chúng kết hợp các kĩ thuật dùng trong tutorial để tạo ra các hiệu ứng ấn tượng hơn. Chúng cũng sử dụng các định dạng dữ liệu được tiền xử lí bởi các tools (FontMaker, ShaderPreprocessor, …). Ngoài ra các chương trình sample còn có giao diện người dùng cho phép xem tỉ lệ khung hình trên giây (FPS), giá trị các tham số và cách tương tác với chương trình.
  + **Demos**: Trong khi sample kết hợp các kĩ thuật dùng trong tutorial để tạo ra các hiệu ứng ấn tượng, còn demo sẽ kết hợp các kĩ thuật dùng trong sample để tạo ra các chương trình hoàn thiện hơn.
  + **Framework**: Đây chính là bộ thư viện mà ta sẽ sử dụng và phát triển để tạo ra game.
  + **Tools**: Là các chương trình hỗ trợ khi sử dụng SDK (Ví dụ như chương trình chuyển đổi định dạng mô hình 3D, chuyển đổi texture, tiền xử lí shader, …).
* **Docs**: Chứa các tài liệu hướng dẫn sử dụng SDK cùng với các thủ thuật khi lập trình đồ họa.

Abc

1. **Các tính năng của Adreno SDK**

Bất cứ ứng dụng OpenGL ES 2.0/3.0 nào cũng cần phải làm nhiều việc hơn là chỉ gọi các hàm để vẽ ra màn hình. Các công việc đó (thường khác nhau trên mỗi nền tảng) bao gồm khởi tạo ứng dụng, kết nối với hệ thống file I/O, cấp phát vùng nhớ, sử dụng thư viện chuẩn. Ngoài ra còn phải load tài nguyên như mô hình 3D, texture và hỗ trợ font chữ, lấy thao tác người dùng (user input), lấy thời gian hệ thống …

Nếu không có một thư viện cơ sở làm các công việc trên, các ứng dụng OpenGL ES sẽ phải tự viết hàng trăm, hàng ngàn dòng mã nguồn lặp lại. Adreno SDK tạo ra một giao diện chung trên tấc cả các nền tảng và nhiều thư viện hỗ trợ giúp các nhà phát triển nhanh chóng tạo ra các ứng dụng OpenGL ES. Quan trọng hơn, nó ngăn cách phần mã nguồn liên quan đến hệ thống/OpenGL ES với phần mã nguồn liên quan đến các kĩ thuật, điều này làm cho người phát triển hứng thú hơn khi lập trình.

Ngoài việc khởi tạo và tương tác với hệ thống, Adreno SDK còn cung cấp bộ thư viện hỗ trợ cho các ứng dụng đồ họa. Ví dụ như OpenGL ES chỉ cung cấp các hàm để vẽ tập các đỉnh chứ không cung cấp thư viện để đọc/quản lí các định dạng mô hình 3D (như FBX, OBJ, COLLADA, …), do đó Adreno SDK cung cấp thư viện để đọc/vẽ mô hình 3D.

1. **Khởi tạo và tương tác với hệ thống**

Để OpenGL ES có thể truy cập được ngữ cảnh vẽ (rendering context) và bề mặt vẽ (drawing surface) của hệ thống, Adreno SDK sử dụng EGL như một cầu nối trung gian. Adreno còn cung cấp một giao diện chung trên mọi nền tảng để truy xuất hệ thống tập tin, thư viện toán học, lấy thao tác người dùng (user input), lấy thời gian hệ thống.

**a.1. Khởi tạo hệ thống**

Adreno SDK sẽ nhận nhiệm vụ khởi tạo hệ thống cho ứng dụng. Mỗi nền tảng đều có hàm Main bắt đầu chương trình (main entry point) được đặt trong tập tin FrmApplication\_Platform.cpp. Ví dụ trên Win32:

|  |
| --- |
| **int** WINAPI WinMain( HINSTANCE, HINSTANCE, LPSTR, **int** )  {  CFrmAppContainer appContainer;  **return** (INT32)appContainer.Run();  } |

Hàm main này sẽ khởi tạo và chạy một đối tượng CFrmAppContainer, đối tượng này lại chứa một đối tượng CFrmApplication, lớp CFrmApplication này là một lớp trừu tượng và cần phải được kế thừa và cài đặt lại trong ứng dụng. Lớp CFrmAppContainer khác nhau trên mỗi nền tảng, lớp này nhận nhiệm vụ khởi tạo cửa sổ màn hình, tương tác với framebuffer để lấy ngữ cảnh vẽ và nhận các sự kiện của chương trình. Lớp kế thừa từ CFrmApplication sẽ khởi tạo cảnh, vẽ cảnh và hủy khi kết thúc.

Một ví dụ của một lớp kế thừa CFrmApplication:

|  |
| --- |
| **class** CSample : **public** CFrmApplication  {  ...  **public**:  CSample( **const** CHAR\* strName );    **virtual** BOOL Initialize();  **virtual** BOOL Resize();  **virtual** VOID Update();  **virtual** VOID Render();  **virtual** VOID Destroy();  }; |

Đối tượng CSample được tạo trong một hàm toàn cục có tên FrmCreateApplicationInstance(). Chương trình ứng dụng cần phải cung cấp bản cài đặt cho hàm này, nếu không sẽ gây ra lỗi khi biên dịch. Một ví dụ về cài đặt hàm này:

|  |
| --- |
| CFrmApplication\* FrmCreateApplicationInstance()  {  **return** **new** CSample( "Sample Name" );  } |

**a.2. Tạo giao diện chung để truy xuất tập tin**

Các lệnh tương tác với hệ điều hành như truy xuất tập tin sẽ được bao bọc lại và tạo ra giao diện chung để đạt được mục tiêu đa nền tảng. Ví dụ như tập tin FrmFile.h chứa một phương thức tên FrmFile\_Open(), đây là phương thức dùng để mở tập tin. Trên Windows, lập trình viên có thể gọi phương thức fopen() thay vì FrmFile\_Open(), nhưng rõ ràng là nếu làm như vậy thì mã nguồn không thể được biên dịch được trên các nền tảng khác. Do vậy việc cung cấp một phương thức chung và cài đặt khác nhau cho mỗi nền tảng sẽ đảm bảo mã nguồn có thể được biên dịch trên tấc cả các nền tảng.

**a.3. Tạo giao diện chung cho thư viện chuẩn C**

Tấc cả các hàm của thư viện stdlib.h cũng được bao bọc lại để đạt mục tiêu đa nền tảng. Những hàm này bao gồm hàm quản lí bộ nhớ FrmMalloc và FrmFree và hàm thao tác với chuỗi trong cstring như FrmStrcpy và FrmSprintf.

**a.4. Lấy thao tác người dùng (user input)**

Adreno SDK cung cấp một tính năng đơn giản để lấy thao tác người dùng (chuột, bàn phím, chạm, …). Trạng thái được nhấn/chạm ở thời điểm hiện tại hay ở vòng lặp game trước có thể được truy vấn thông qua hàm FrmGetInput().

Ví dụ:

|  |
| --- |
| VOID CSample::Update()  {  UINT32 nButtons;  UINT32 nPressedButtons;    FrmGetInput( &m\_Input, &nButtons, &nPressedButtons );  **if**( nButtons & INPUT\_KEY\_0 )  {  // Do something if the '0' key is pressed  }  } |

**a.5. Thư viện toán học 2D/3D**

FrmMath.h chứa nhiều struct và các phương thức toán học bao gồm vector và ma trận. Kiểu được dùng nhiều nhất là FRMVECTOR3, FRMVECTOR4 và FRMMATRIX4X4.

1. **Thư viện hỗ trợ OpenGL ES**

Adreno SDK cung cấp cho lập trình viên thư viện để biên dịch shader, quản lí vertex buffer, index buffer, quản lí đối tượng texture.

**b.1. Hỗ trợ shader GLSL**

SDK cung cấp thư viện để đọc, biên dịch, liên kết shader thông qua một hàm duy nhất FrmCompileShaderProgramFromFile(). Thêm vào đó, hàm này còn cho phép người dùng truyền vào danh sách các FRM\_SHADER\_ATTRIBUTE để ràng buộc các thuộc tính của đỉnh trong ứng dụng và shader. Sau khi biên dịch xong, chúng ta có thể trích xuất ra địa chỉ biến uniform, dùng shader để vẽ, … Một ví dụ:

|  |
| --- |
| FRM\_SHADER\_ATTRIBUTE pShaderAttributes[] =  {  { "In\_Position", FRM\_VERTEX\_POSITION },  { "In\_Normal", FRM\_VERTEX\_NORMAL },  { "In\_TexCoord", FRM\_VERTEX\_TEXCOORD0 },  };  GLuint hShader;  **if**( FALSE == FrmCompileShaderProgramFromFile(  "Media\\Shaders\\Shader.vs",  "Media\\Shaders\\Shader.fs",  &hShader,  pShaderAttributes, 3 ) )  **return** FALSE;  // Make a record of the location for each shader constant  slotWorldMat = glGetUniformLocation( hShader, "matWorld" );  .... |

1. **Cung cấp định dạng mô hình 3D**

Một định dạng mô hình 3D được dùng trong game phải thỏa mãn 2 tiêu chí:

* Dung lượng nhẹ
* Đọc và xử lí nhanh

Các định dạng phổ biến (interchange format) như FBX, OBJ, COLLADA, … không thỏa mãn 2 tiêu chí này.

FBX thì chứa quá nhiều thứ không được dùng trong game làm cho dung lượng tập tin lớn và cấu trúc định dạng này quá phức tạp nên việc xử lí sẽ rất chậm. OBJ thì không có dạng nhị phân nên dung lượng cũng lớn. COLLADA dùng XML thì càng không thể chấp nhận được, …

Do đó Adreno SDK cung cấp một định dạng lưu trữ mô hình 3D thỏa mãn các tiêu chí trên. Đây là tập tin dạng nhị phân, được thiết kế sao cho việc đọc và xử lí nhanh nhất có thể. Tập tin này được chia làm 2 phần: Một phần dùng để lưu dữ liệu tồn tại trong suốt quá trình sống của mô hình và phần còn lại lưu dữ liệu có thể bỏ đi sau khi load xong mô hình.

Việc load một tập tin mô hình chỉ cần 2 lần cấp phát bộ nhớ và 2 lần đọc tập tin. Sau đó là đến bước “pointer fixup”, ở bước này chương trình sẽ đổ dữ liệu cho cấu trúc dữ liệu của mô hình. Tiếp theo là bước tạo và đổ dữ liệu cho vertex buffer và index buffer. Cuối cùng là gắn texture cho các sub-mesh trong mô hình.

|  |
| --- |
| CFrmMesh Mesh;  **if**( FALSE == Mesh.Load( "Media\\Meshes\\Test.mesh" ) )  **return** FALSE;  **if**( FALSE == Mesh.MakeDrawable( &resource ) )  **return** FALSE; |

Một đối tượng CFrmMesh chứa một danh sách các đối tượng FRM\_MESH\_FRAME (khung) dạng cây. Mỗi khung chứa dữ liệu cho mô hình và con trỏ trỏ tới khung con và khung anh/chị. Dữ liệu cho mô hình được lưu trữ trong cấu trúc FRM\_MESH (chứa dữ liệu về đỉnh, tam giác của mô hình) và FRM\_MESH\_SUBSET (chứa dữ liệu về vật liệu và texture của mô hình).

Mỗi khung còn chứa dữ liệu cho chuyển động nhân vật (animation) được lưu trong cấu trúc FRM\_ANIMATION. Dữ liệu chuyển động là một tập các khóa (key-frame), mỗi khóa thực sự là giá trị vị trí, góc quay, kích cỡ của từng bộ phận của mô hình. Để chuyển động mô hình, ta dùng giá trị thời gian tại thời điểm hiện tại để nội suy giá trị vị trí, góc quay, kích cỡ của từng bộ phận, sau đó xây dựng ma trận biến hình dựa trên các giá trị đó, và ma trận này sẽ tạo ra tư thế cho bộ phận đó. Bởi vì một mô hình có thể có nhiều chuyển động (đi, chạy, nhảy, …), nên mỗi chuyển động sẽ được lưu trong từng tập tin khác nhau.

1. **Cung cấp định dạng texture**

Việc đọc và xử lí các định dạng ảnh quen thuộc như TGA, BMP, JPG có nhiều nhược điểm: Mã nguồn rất phức tạp và chạy chậm vì xử lí rất nhiều. Ngoài ra, đối với các loại ảnh nén, việc giải nén trên thiết bị di động có thể mất vài giây cho mỗi ảnh.

Do đó, một giải pháp cho vấn đề trên là ta sẽ xử lí ảnh từ trước, nén và lưu trữ vào một định dạng do chúng ta tự xây dựng, làm cho việc đọc/xử lí trở nên đơn giản và nhanh hơn. Thêm vào đó, định dạng này còn có khả năng gộp nhiều ảnh vào trong một tập tin, làm cho việc đọc tập tin nhanh hơn và dễ dàng quản lí khi có rất nhiều ảnh.

Để hỗ trợ cho định dạng ảnh này, Adreno SDK bao gồm một công cụ có tên ResourcePacker. Đầu vào của công cụ này là một tập tin XML mô tả các tập tin ảnh sẽ xử lí và cách xử lí. Đầu ra sẽ là tập tin \*.pak duy nhất. Tập tin này sẽ được đọc và xử lí thông qua lớp CFrmPackedResource. Mỗi texture trong tập tin \*.pak phải có một ID (là chuỗi ASCII) để có thể được trích xuất ra vào trong một lớp có tên CFrmTexture.

|  |
| --- |
| CFrmPackedResource resource;  **if**( FALSE == resource.LoadFromFile( "Media\\Textures.pak" ) )  **return** FALSE;  ....  CFrmTexture\* pLogoTexture = resource.GetTexture( "Logo" ); |

1. **Thư viện giao diện người dùng**

Adreno SDK chỉ cung cấp thư viện để vẽ chữ và texture, trong khi trong game yêu cầu nhiều đối tượng phức tạp hơn (như nút bấm, nút chọn, danh sách…), nên phần giao diện người dùng này cần phải được phát triển thêm.

1. **Thư viện font chữ**

Adreno SDK hỗ trợ font dạng rasterized, tức là mỗi kí tự (glyph) là một bức ảnh, tấc cả các bức ảnh này sẽ được gộp lại thành một bức ảnh duy nhất (sheet of glyphs). Adreno SDK bao gồm một công cụ có tên FontMaker, công cụ này có nhiệm vụ chuyển đổi font của hệ điều hành (dạng truetype) thành tập tin ảnh (chính là sheet of glyphs). Đầu ra của FontMaker là 3 tập tin: Tập tin ảnh TGA (là ảnh của tấc cả các kí tự), tập tin dữ liệu font (chứa thông tin về tọa độ cũng như khoảng cách giữa các kí tự) và một tập tin XML (đây là đầu vào của ResourcePacker để tạo ra tập tin \*.pak).

Cũng giống như texture, font chữ cũng được lưu trữ trong tập tin \*.pak. Và được xử lí bởi lớp CFrmFont. Sau đó, chữ sẽ được vẽ ra màn hình thông qua hàm DrawText:

|  |
| --- |
| CFrmFont Font;  BOOL bResult = Font.Create( "Media\\Fonts\\SomeFont16.pak" );  ...  Font.DrawText( 10, 10, FRMCOLOR\_WHITE, "Draw this text." ); |

1. **Các công cụ hỗ trợ**

**g.1. Trình giả lập OpenGL ES**

Trình giả lập này được dùng để chạy ứng dụng OpenGL ES trên desktop. Mục đích là để hỗ trợ lập trình viên thiết kế game OpenGL ES trên desktop bởi vì chạy trên môi trường desktop có nhiều thuận lợi như công cụ debug mạnh mẽ chẳng hạn.

**g.2. Chuyển đổi mô hình 3D**

Đây là một công cụ dùng để chuyển đổi mô hình 3D dạng FBX sang định dạng mô hình của Adreno SDK.

**g.3. Biên dịch shader ra dạng nhị phân**

Thường thì shader được biên dịch lúc runtime, điều này có thể làm chậm chương trình do việc biên dịch shader ra dạng nhị phân có thể tốn kém. Do vậy việc biên dịch shader trước khi chạy làm tăng tốc chương trình.

**g.4. FontMaker**

Công cụ này dùng để chuyển đổi font chữ hệ thống sang font chữ của Adreno SDK.

**g.5. ResourcePacker**

Công cụ này dùng để chuyển đổi định dạng ảnh phổ biến (JPG, TGA, …) sang định dạng texture của Adreno SDK.

**CHƯƠNG XXX. Một số vấn đề và giải pháp khi xây dựng game trên Adreno SDK**

1. **Đọc và xử lí mô hình 3D**

* **Vấn đề 1**: Dùng định dạng mô hình 3D nào?

Việc chọn lựa một định dạng mô hình 3D thích hợp để sử dụng là rất quan trọng, nó không chỉ ảnh hưởng đến tốc độ game, mà còn phải xem xét đến nhiều yếu tố như: Nguồn mô hình định dạng đó trên mạng có phong phú hay không, có thư viện tốt để đọc định dạng đó hay không, …

* **Giải pháp**:

Các định dạng phổ biến như FBX, OBJ, COLLADA không thích hợp dùng trong game, nên ta phải tạo ra định dạng mô hình riêng.

Adreno SDK cung cấp 2 định dạng mô hình như vậy: Mô hình 3D loại 1 có đuôi \*.mesh và mô hình 3D loại 2 có đuôi \*.model. Ưu điểm của mô hình loại 1 là Adreno cung cấp sẵn một lớp có tên CfrmMesh, lớp này sẽ tự đọc và vẽ mô hình 3D, do đó việc dùng mô hình loại 1 là rất dễ dàng. Còn đối với mô hình loại 2, Adreno SDK chỉ đọc dữ liệu, việc sử dụng dữ liệu đó để vẽ như thế nào do người dùng tự làm.

Như vậy ta thấy dùng loại 1 thì dễ dàng nhưng không thể tùy biến được, dùng loại 2 rất vất vả nhưng chúng ta làm chủ hoàn toàn quá trình vẽ mô hình, do đó ta có thể thêm các bước xử lí để thỏa mãn hiệu ứng mong muốn.

Trong quá trình làm game, thường nảy sinh các yêu cầu như bỏ bớt số lượng bone index của một đỉnh. Nếu dùng định dạng loại 1 thì không thể đáp ứng yêu cầu này được vì ta không thể can thiệp vào quá trình đọc và vẽ mô hình. Do đó nhóm chúng em quyết định sử dụng định dạng mô hình loại 2.

* **Vấn đề 2**: Đọc và vẽ mô hình 3D
* **Giải pháp**:

Do Adreno SDK không cung cấp lớp đối tượng để quản lí mô hình loại 2, nên ta sẽ xây dựng lớp FileMesh1 để quản lí việc đọc/vẽ mô hình:

|  |
| --- |
| **class** FileMesh1 : **public** Mesh  {  **public**:  **virtual** **void** init(  FileMesh1::MeshData& model,  FileMesh1::MeshTextures& modelTexture,  Shader& shader,  Material\* material = **nullptr**);  **virtual** **void** update(Timer& timer);  **virtual** **void** render(Camera& camera, Light\* light = **nullptr**);  **protected**:  **void** prepareRenderSubmesh(**int** submesh);  **protected**:  Adreno::Model\* m\_model;  Texture\*\* m\_modelTexture;  GLuint\* m\_vertexBuffer;  GLuint\* m\_indexBuffer;  VERTEX\_FORMAT\_MAP\* m\_vertexFormatMap;  }; |

Ta sẽ không đọc mô hình bên trong lớp FileMesh1, mà đọc bên ngoài thông qua hàm FrmLoadModelFromFile(), sau đó truyền dữ liệu mô hình cho lớp FileMesh1. Ngoài dữ liệu mô hình, lớp FileMesh1 còn yêu cầu truyền texture, shader và vật liệu. Hàm init() có nhiệm vụ khởi tạo vertex buffer, index buffer và nhiều thứ khác để việc vẽ mô hình có thể được tiến hành.

Sau khi khởi tạo mô hình xong, ta có thể vẽ mô hình ra màn hình. Hàm render() có nhiệm vụ lặp qua từng bộ phận (sub-mesh) của mô hình, sau đó gán giá trị texture, ma trận biến hình, giá trị ánh sáng, giá trị vật liệu, … ứng với bộ phận đó cho shader, cuối cùng là gọi hàm vẽ để vẽ bộ phận đó ra màn hình.

* **Vấn đề 3**: Vẽ một mô hình 3D nhiều lần (instancing)

Một tính năng cần phải có đó là vẽ cùng một mô hình nhiều lần ra màn hình. Ví dụ như một rừng cây gồm nhiều cây (mỗi cây chỉ khác nhau về vị trí, kích thước, màu sắc còn về hình dạng thì không đổi), hoặc hoặc bầy quái gồm nhiều con quái chỉ khác về vị trí, kích thước, tư thế, …

* **Giải pháp**:

Một giải pháp đơn giản nhất là mỗi thực thể (instance) ứng với một đối tượng FileMesh1, lúc vẽ ta chỉ cần duyệt qua hết các đối tượng này và gọi hàm vẽ. Nhưng cách này có 2 nhược điểm:

* Tốn bộ nhớ: Mỗi đối tượng FileMesh1 sẽ chứa dữ liệu mô hình (tập đỉnh, tam giác, …) giống nhau. Điều này sẽ gây lãng phí bộ nhớ rất lớn.
* Chạy chậm: Khi chúng ta vẽ đối tượng FileMesh1 thứ nhất, OpenGL ES sẽ sao chép tập đỉnh vào vertex buffer để vẽ, sau đó thiết lập nhiều điều kiện để việc vẽ có thể thực hiện. Đến khi vẽ đối tượng 2, OpenGL ES cũng lặp lại các bước như trên.

OpenGL ES đề nghị 3 phương pháp cho vẽ một mô hình nhiều lần:

* Thực hiện trên GPU (hardware instancing): Việc vẽ nhiều lần này sẽ do GPU thực hiện thông qua lệnh glDrawArraysInstanced(). Nhưng phương pháp này chỉ thực hiện được trên OpenGL ES 3.0.
* Thực hiện trên CPU (software instancing using single drawcall): Thay vì chúng ta truyền n lần một tập đỉnh giống nhau vào GPU thì chúng ta sẽ truyền một tập có kích thước gấp n lần tập đỉnh ban đầu vào GPU. Mỗi đỉnh sẽ có số thứ tự (ID) để biết đỉnh đó thuộc mô hình nào. Lúc vẽ ta dựa vào ID để xác định vị trí, kích cỡ, màu sắc của đỉnh đó. Nhược điểm của cách này là số lượng mô hình được vẽ phải cố định.
* Thực hiện trên CPU (software instancing using multiple drawcall): Trước khi vẽ, ta sao chép tập đỉnh vào vertex buffer, sau đó thiết các trạng thái vẽ, cuối cùng là gọi nhiều lần lệnh vẽ.

Nhóm chúng em quyết định sử dụng cách thứ ba vì số lượng mô hình để vẽ là thay đổi và game đang viết sử dụng OpenGL ES 2.0.

|  |
| --- |
| **class** Mesh  {  **public**:  **struct** Instance  {  MyVec3 Position;  MyVec3 Rotation;  MyVec3 Scale;  MyMat4 World;  **bool** Visible;  };  **public**:  **virtual** **void** init(Shader& shader, Material\* material = **nullptr**);  **virtual** **void** update(Timer& timer);  **virtual** **void** render(Camera& camera, Light\* light = **nullptr**);  // Getter  **int** getNumInstances();  Instance\* getInstance(**int** id);  // An instance must be initialized on heap (i.e. using 'new' operator)  **void** addInstance(Instance\* instance);  **void** removeInstance(Instance\* instance);  **protected**:  Shader\* m\_shader;  std::vector<Instance\*> m\_instances;  }; |

1. **Chuyển động mô hình 3D (Animation)**

* **Vấn đề 1**: Nguyên lý của chuyển động mô hình 3D

Trước khi tìm hiểu cách đọc và xử lí chuyển động, ta nên tìm hiểu làm thế nào để tạo chuyển động cho mô hình 3D. Việc hiểu rõ nguyên lí hoạt động giúp ta có thể cấu hình, thêm các tính năng mới.

* **Giải pháp**:

Để tạo chuyển động cho mô hình 3D, Adreno SDK sử dụng kĩ thuật gọi là “skinned mesh”. Trong kĩ thuật này, mô hình 3D được gắn với một hệ thống khung xương (skeleton), hệ thống này gồm nhiều xương (bone) được tổ chứa thành cây, ta hiểu nôm na xương là một điểm trong không gian ba chiều. Một đỉnh trên mô hình 3D sẽ chịu ảnh hưởng bởi một hay nhiều xương, số lượng xương ảnh hưởng càng lớn thì chuyển động sẽ càng mượt hơn. Ví dụ như các đỉnh nằm xung quanh cùi chỏ sẽ bị ảnh hưởng bởi xương bắp tay và xương cẳng tay với các hệ số ảnh hưởng thích hợp để không tạo nên đường đứt gãy tại cùi chỏ lúc vẽ.

Mỗi xương sẽ có một ma trận để xác định vị trí, góc quay và kích cỡ của xương đó. Như vậy một hệ thống khung xương sẽ tạo thành một chuỗi ma trận (matrix palette). Lúc vẽ mô hình, đối với mô hình tĩnh ta chỉ cần truyền một ma trận biến hình, ma trận này sẽ áp dụng lên tấc cả các đỉnh, còn đối với mô hình có chuyển động ta sẽ truyền một chuỗi ma trận của khung xương, ma trận của xương nào thì sẽ áp dụng lên các đỉnh bị ảnh hưởng bởi xương đó. Với một giá trị của chuỗi ma trận sẽ tạo ra một tư thế cho mô hình. Do đó ta liên tiếp điều chỉnh giá trị của chuỗi ma trận, thì các tư thế của mô hình cũng sẽ thay đổi một cách liên tục và chính điều này làm nên chuyển động cho mô hình.

Đối với đỉnh của mô hình động, ngoài giá trị vị trí, tọa độ texture, vector normal như mô hình tĩnh còn có thêm 4-số nguyên biểu thị chỉ số của các xương ảnh hưởng đến (bone indices) và 4-số thực biểu thị mức độ ảnh hưởng (bone weights). Số 4 có nghĩa là mỗi đỉnh bị ảnh hưởng bởi tối đa 4 xương. Có thể có nhiều xương ảnh hưởng hơn, nhưng thực tế chỉ 2-3 xương ảnh hưởng là đủ, nhiều quá không có nhiều tác dụng mà chỉ làm chậm ứng dụng.

Một ví dụ về khai báo đỉnh:

|  |
| --- |
| **struct** SKINNED\_VERTEX  {  FRMVECTOR3 vPosition;  BYTE BoneIndices[4];  FLOAT32 BoneWeights[4];  FRMVECTOR3 vNormal;  FRMVECTOR3 vTexCoord;  }; |

Trong vertex shader, giá trị vị trí và vector normal bị biến hình nhiều lần bởi các ma trận của xương ảnh hưởng. Giá trị sau khi nhân với ma trận xương sẽ tiếp tục nhân với giá trị mức độ ảnh hưởng, sau đó cộng dồn tấc cả các giá trị lại ta sẽ có giá trị vị trí, vector normal sẽ được vẽ.

Trong Adreno SDK dữ liệu chuyển động và dữ liệu đỉnh của mô hình được đặt trong 2 tập tin riêng biệt. Ưu điểm của việc này đó là một dữ liệu chuyển động có thể được áp dụng vào nhiều mô hình khác nhau. Ta đã biết dữ liệu mô hình gồm một hệ thống khung xương, mỗi xương có một tên (ID). Còn dữ liệu chuyển động chứa giá trị ma trận của xương, mỗi xương trong dữ liệu chuyển động cũng cần phải có tên. Và một khi tấc cả các tên trong hai dữ liệu này hợp nhau, khi đó dữ liệu chuyển động có thể được áp dụng lên mô hình.

Dữ liệu chuyển động là một tập của chuỗi ma trận (key-frame), mỗi ma trận cho biết vị trí, góc quay, kích cỡ của xương tại một thời điểm nào đó. Trong suốt quá trình hoạt động của game, hệ thống quản lí chuyển động sẽ cung cấp giá trị thời gian hiện tại, giá trị này được dùng để nội suy ra ma trận của xương. Những ma trận này sau đó được dùng để xây dựng chuỗi ma trận và truyền vào vertex shader.

* **Vấn đề 2**: Đọc và xử lí tập tin chuyển động
* **Giải pháp**:

Để quản lí mô hình động, ta xây dựng lớp SkinnedMesh1:

|  |
| --- |
| **class** SkinnedMesh1 : **public** FileMesh1  {  **public**:  **static** **const** UINT32 MAX\_BONES = 60;  **static** **const** UINT32 TOTAL\_BONES = 256;  **static** **const** UINT32 TICKS\_PER\_FRAME = 150;  **public**:  // For an anim action (idle, walk, attack, ...), at a specific time  // (e.g. the end of action), a event  // will be trigger.  // A pointer to IOnPerformAActListener will listen that event.  **class** IOnPerformAActListener  {  **public**:  **virtual** **void** OnPerformAAct(**void**\* tag) = 0;  };  // An action (e.g. run, walk, dead, ...) will begin at Frame of  // FrameStart and last FrameLength frames.  **struct** AnimAction  {  UINT32 FrameStart;  UINT32 FrameLength;  };  **struct** Instance : **public** Mesh::Instance  {  // At time of t0, we're being between m\_leftFrame Frame and  // m\_rightFrame Frame.  // We're interpolating transform at t0 basing on m\_leftFrame  // Frame and m\_rightFrame Frame.  INT32 LeftFrame;  INT32 RightFrame;  FLOAT32 FrameWeight;  **bool** LoopedAction;  MyString CurrentAction;  MyString NextAction;  UINT32 TotalTicks;  };  **public**:  **void** init(  FileMesh1::MeshData& model,  SkinnedMesh1::AnimData& anim,  FileMesh1::MeshTextures& modelTexture,  Shader& shader,  Material\* material = **nullptr**,  FLOAT32 speedFactor = 1.0f);  **void** update(Timer& timer);  **void** render(Camera& camera, Light\* light = **nullptr**);  **private**:  Adreno::Animation\* m\_anim;  // Bone remapping  UINT32 m\_boneRemap[MAX\_BONES];  UINT32 m\_boneRemapCount;  // Animation speed and range  FLOAT32 m\_speedFactor;  std::map<MyString, AnimAction> m\_animActions;  }; |

Ta sẽ đọc dữ liệu từ bên ngoài rồi truyền vào cho lớp SkinnedMesh1, điều này giúp ta có thể áp dụng một dữ liệu chuyển động lên nhiều mô hình một cách hiệu quả.

Trước khi vẽ mô hình động, ta cần phải tính toán chuỗi ma trận cho khung xương:

|  |
| --- |
| **void** SkinnedMesh1::setWorldArray(SkinnedMesh1::Instance\* instance)  {  // Prepare this frame's transforms for each of the bones  FRMMATRIX4X3 matWorldMatrixArray[MAX\_BONES];  **for** (UINT32 boneIndex = 0; boneIndex < m\_boneRemapCount; ++boneIndex)  {  FRMMATRIX4X4 matBoneMatrix = FrmMatrixIdentity();  // Get the current and last transforms  Adreno::Joint\* pJoint = m\_model->Joints + m\_boneRemap[boneIndex];  Adreno::AnimationTrack\* pTrack =  m\_anim->Tracks + m\_boneRemap[boneIndex];  Adreno::Transform\* pCurTransform;  Adreno::Transform\* pLastTransform;  **if** (pTrack->NumKeyframes >= 0)  {  pCurTransform = pTrack->Keyframes + instance->RightFrame;  pLastTransform = pTrack->Keyframes + instance->LeftFrame;  }  **else**  {  pCurTransform = &pJoint->Transform;  pLastTransform = &pJoint->Transform;  }  // Interpolate between the last and current transforms  Adreno::Transform transform;  transform.Position = FrmVector3Lerp(  pLastTransform->Position,  pCurTransform->Position,  instance->FrameWeight);  transform.Rotation = FrmVector4SLerp(  pLastTransform->Rotation,  pCurTransform->Rotation,  instance->FrameWeight);  // Apply inverse bind transform to final matrix  FRMMATRIX4X4 matInverseBindPosition =  FrmMatrixTranslate(pJoint->InverseBindPose.Position);  FRMMATRIX4X4 matInverseBindRotation =  FrmMatrixRotate(pJoint->InverseBindPose.Rotation);  matBoneMatrix = matInverseBindRotation;  matBoneMatrix = FrmMatrixMultiply(  matBoneMatrix,  matInverseBindPosition);  // Apply interpolated transform to final matrix  FRMMATRIX4X4 matBonePosition =  FrmMatrixTranslate(transform.Position);  FRMMATRIX4X4 matBoneRotation =  FrmMatrixRotate(transform.Rotation);  matBoneMatrix = FrmMatrixMultiply(matBoneMatrix, matBoneRotation);  matBoneMatrix = FrmMatrixMultiply(matBoneMatrix, matBonePosition);  // Place it in a 4x3 matrix  matBoneMatrix = FrmMatrixTranspose(matBoneMatrix);  memcpy(  &matWorldMatrixArray[boneIndex],  &matBoneMatrix,  **sizeof**(FRMMATRIX4X3));  }  // Set bone transforms  m\_shader->setUniform(  "u\_worldArray",  (FLOAT32\*)&matWorldMatrixArray[0],  MAX\_BONES \* 3,  4);  } |

Như chúng ta thấy, giá trị chuỗi ma trận tại thời điểm t được nội suy từ giá trị hai chuỗi ma trận tại thời điểm t1 và t2, trong đó t1 < t < t2. Sau khi tính toán xong, chuỗi ma trận này sẽ được truyền vào vertex shader để vẽ mô hình.

* **Vấn đề 3**: Tối ưu hóa

Việc xử lí mô hình chuyển động là một công việc gây tốn bộ nhớ và tốn nhiều chi phí. Nếu chúng ta không tối ưu hóa một vài bước, có thể làm cho ứng dụng trở nên chậm chạp và thậm chí có thể gây văng chương trình lúc chạy.

* **Giải pháp**:

Một vài thủ thuật sau đây có thể giúp tối ưu ứng dụng dùng mô hình chuyển động:

* Loại bỏ xương không ảnh hưởng đến bất cứ đỉnh nào:

Số lượng xương tối đa mà Adreno SDK hỗ trợ là hằng số cố định (60 xương). Nếu mô hình có số xương nhiều hơn số lượng này, Adreno SDK không thể vẽ mô hình này được. Ngoài ra số lượng xương càng nhỏ thì chi phí tính toán cũng sẽ ít hơn. Do vậy việc giảm đi số lượng xương trong mô hình là rất quan trọng.

Hàm sau đây sẽ loại bỏ những xương không ảnh hưởng đến bất kì đỉnh nào trong mô hình:

|  |
| --- |
| BOOL SkinnedMesh1::RemapBoneIndices( Adreno::Model\* pModel,  Adreno::Animation\* pAnim )  {  // Return false if number of total bones is  // greater than supported  **if** ( pModel->NumJoints > TOTAL\_BONES )  {  **return** FALSE;  }  // Temporary buffer  UINT8 processedBone[TOTAL\_BONES];  memset( processedBone, 0xFF, **sizeof**( processedBone ) );  // The number of bones remapped  m\_boneRemapCount = 0;  // Process each mesh  **for** ( INT32 meshIndex = 0; meshIndex < m\_pModel->NumMeshes;  ++meshIndex )  {  Adreno::Mesh\* pMesh = pModel->Meshes + meshIndex;  // Find the index for skinindex and skinweight properties  INT32 skinIndexProperty = GetPropertyIndexFromName( pMesh,  "skinindex" );  INT32 skinWeightProperty = GetPropertyIndexFromName( pMesh,  "skinweight" );  **if** ( skinIndexProperty < 0 || skinWeightProperty < 0 )  {  **continue**;  }  // The number of bones that can influence a vertex  UINT32 numberOfBones =  pMesh->Vertices.Format.  Properties[skinIndexProperty].NumValues();  // Look at each vertex of the mesh  **for** (  UINT32 vertexIndex = 0, vertexOffset = 0;  vertexIndex < pMesh->Vertices.NumVerts;  ++vertexIndex,  vertexOffset += pMesh->Vertices.Format.Stride )  {  UINT32\* pBoneIndex = ( UINT32\* ) (  pMesh->Vertices.Buffer +  vertexOffset +  pMesh->Vertices.Format.  Properties[skinIndexProperty].Offset );  FLOAT32\* pBoneWeight = ( FLOAT32\* ) (  pMesh->Vertices.Buffer +  vertexOffset +  pMesh->Vertices.Format.  Properties[skinWeightProperty].Offset );  // Add each bone that transform vertices to the table  **for** ( UINT32 boneCount = 0; boneCount < numberOfBones;  ++boneCount )  {  UINT32 boneIndex = \*( pBoneIndex + boneCount );  FLOAT32 boneWeight = \*( pBoneWeight + boneCount );  // Ignore bones that have no influence  // (weight is zero)  **if** ( boneWeight != 0.0f )  {  **if** ( processedBone[boneIndex] == 0xFF )  {  **if** ( m\_boneRemapCount >= MAX\_BONES )  {  **return** FALSE;  }    processedBone[boneIndex] =  m\_boneRemapCount;  m\_boneRemap[m\_boneRemapCount++] =  boneIndex;  }    \*( pBoneIndex + boneCount ) =  processedBone[boneIndex];  }  }  }  }  **return** TRUE;  } |

* Hạn chế số lượng xương ảnh hưởng lên một đỉnh:

Số lượng xương ảnh hưởng lên một đỉnh càng ít thì chi phí tính toán sẽ càng thấp, do đó chúng em quyết định mỗi đỉnh chỉ bị ảnh hưởng bởi một xương, điều này sẽ tối thiểu hóa chi phí tính toán.

|  |
| --- |
| // Get rid of unused bones through remapping  **static** BOOL RemapBoneIndices(Adreno::Model\* pModel)  {  // Process each mesh  **for** (INT32 meshIndex = 0; meshIndex < pModel->NumMeshes; ++meshIndex)  {  Adreno::Mesh\* pMesh = pModel->Meshes + meshIndex;  ...    // The number of bones that can influence a vertex  UINT32 numberOfBones =  pMesh->Vertices.Format.  Properties[skinIndexProperty].NumValues();  // Look at each vertex of the mesh  **for** (UINT32 vertexIndex = 0;  vertexIndex < pMesh->Vertices.NumVerts;  ++vertexIndex)  {  ...  // Add each bone that transform vertices to the table  FLOAT32 maxBoneWeight = -1;  UINT32 maxBoneIndex = 0;  **for** (UINT32 boneCount = 0; boneCount < numberOfBones;  ++boneCount)  {  UINT32 boneIndex = \*(pBoneIndex + boneCount);  FLOAT32 boneWeight = \*(pBoneWeight + boneCount);  // Determine max bone weight  **if** (boneWeight > maxBoneWeight)  {  maxBoneWeight = boneWeight;  maxBoneIndex = \*(pBoneIndex + boneCount);  }  }  // It's only support that a vertex is  // affected by a bone  (\*pBoneWeight) = 1.0;  (\*pBoneIndex) = maxBoneIndex;  }  }  **return** TRUE;  } |

Như ta thấy ở hàm trên, xương duy nhất sẽ ảnh hưởng lên đỉnh sẽ là xương có mức độ ảnh hưởng lớn nhất.

* Dùng cấu trúc dữ liệu tối ưu để lưu trữ chuỗi ma trận khung xương:

Ta thấy chuỗi ma trận là một mảng của ma trận 4x4. Nhưng ở mỗi ma trận 4x4 thì cột cuối cùng không bao giờ được sử dụng, do đó dùng mảng ma trận 4x3 sẽ tốt hơn. Nhưng shader GLSL lại không hỗ trợ ma trận 4x3, do đó thay vì lưu một mảng các ma trận 4x3, ta sẽ lưu một mảng các vector4 (ma trận 4x3 sẽ được tách thành 3 vector4).

* **Vấn đề 4**: Một mô hình có nhiều động tác

Một mô hình có thể có nhiều động tác (đứng yên, chạy, đánh, …). Mỗi động tác này sẽ được lưu trữ trong một tập tin dữ liệu chuyển động.

* **Giải pháp**:

Với mỗi tập tin chứa dữ liệu chuyển động, ta sẽ lưu tạm vào một cấu trúc dữ liệu. Sau đó ta sẽ gộp tấc cả các dữ liệu chuyển động vào một cấu trúc dữ liệu duy nhất. Mỗi chuyển động sẽ có một tên (ID) và địa chỉ trong cấu trúc dữ liệu trên để có thể được trích xuất sau này.

|  |
| --- |
| **class** SkinnedMesh1 : **public** FileMesh1  {  **public**:  // An action (e.g. run, walk, dead, ...) will begin at  // Frame of FrameStart and last FrameLength frames.  **struct** AnimAction  {  UINT32 FrameStart;  UINT32 FrameLength;  };  **public**:  **void** init(  FileMesh1::MeshData& model,  SkinnedMesh1::AnimData& anim,  FileMesh1::MeshTextures& modelTexture,  Shader& shader,  Material\* material = **nullptr**,  FLOAT32 speedFactor = 1.0f);  **void** update(Timer& timer);  **void** render(Camera& camera, Light\* light = **nullptr**);  **private**:  ...  // Container that contains all animation data  std::map<MyString, AnimAction> m\_animActions;  }; |

1. **Tạo địa hình**

Địa hình là một thành phần không thiếu trong tấc các game, nó đóng vai trò như mặt đất để đặt các công trình và là nơi để nhân vật di chuyển.

* **Vấn đề 1**: Xây dựng mô hình cho địa hình

Địa hình giống như các mô hình 3D khác, cũng được tạo thành từ các tam giác. Chỉ có điều nó không được đọc từ tập tin mà chính chúng ta phải phát sinh các đỉnh cho nó (procedural model).

Có hai loại địa hình:

* Địa hình phẳng (flat terrain): Địa hình chỉ là một hình chữ nhật phẳng.
* Địa hình gồ ghề (rough terrain): Địa hình là một lưới các đỉnh, trong đó các đỉnh có độ cao khác nhau tạo ra chỗ lồi , chỗ lõm. Để xác định độ cao cho từng đỉnh, ta thường dùng một texture (height map).

Do yêu cầu của game không cần địa hình gồ ghề, nên chúng em sử dụng địa hình phẳng trong game.

* **Giải pháp**:

Để quản lí và vẽ địa hình, ta xây dựng lớp FlatTerrain:

|  |
| --- |
| **struct** FlatTerrainProperties  {  // TileRepeatFactor.x: Tile factor for diffuse texture 1  // TileRepeatFactor.y: Tile factor for diffuse texture 2  // TileRepeatFactor.z: Tile factor for blending map  MyVec3 TileRepeatFactor;  // Threshold to begin blending:  // - if 0 <= blending\_value < Threshold.x: No blending at all,  // take diffuse texture 1  // - if Threshold.y < blending\_value <= 1: No blending at all,  // take diffuse texture 2  // - otherwise, blend using linear interpolation  MyVec2 Threshold;  };  // Flat terrain is a flat-square quad having properties:  // - Perpendicular to Oy-axis at y=0  // - Aligned with Ox, Oz, axis  **class** FlatTerrain : **public** OnPressListenee  {  **public**:  **void** init(  Shader& shader,  Texture& diffuseMap1,  Texture& diffuseMap2,  Texture& blendMap,  **const** MyVec3& center,  **const** MyVec2& size,  FlatTerrainProperties& properties);  **void** update(Timer& timer, **bool** isPressed, MyVec3& pressedPoint);  **void** render(Camera& camera);  **private**:  BasicMesh m\_mesh;  Texture\* m\_diffuseMap1;  Texture\* m\_diffuseMap2;  Texture\* m\_blendMap;  FlatTerrainProperties m\_properties;  MyVec3 m\_center;  MyVec2 m\_size;  }; |

Mô hình cho địa hình là một hình chữ nhật 4 đỉnh:

|  |
| --- |
| std::vector<PosTexVertex> vertices;  vertices.resize(4);  vertices[0] = PosTexVertex(MyVec3(-0.5f, +0.0f, -0.5f), MyVec2(0, 0));  vertices[1] = PosTexVertex(MyVec3(-0.5f, +0.0f, +0.5f), MyVec2(0, 1));  vertices[2] = PosTexVertex(MyVec3(+0.5f, +0.0f, +0.5f), MyVec2(1, 1));  vertices[3] = PosTexVertex(MyVec3(+0.5f, +0.0f, -0.5f), MyVec2(1, 0));  UIntArray indices;  indices.resize(6);  indices[0] = 0;  indices[1] = 1;  indices[2] = 2;  indices[3] = 0;  indices[4] = 2;  indices[5] = 3; |

* **Vấn đề 2**: Phủ texture lên địa hình

Để tạo nên sự sinh động cho game, địa hình cần được phủ texture (chỗ thì có cỏ, chỗ thì có đất, đá, …).

* **Giải pháp**:

Ta sẽ dùng kĩ thuật multi-texture để phủ nhiều loại texture lên địa hình. Trong kĩ thuật này, để xác định loại texture để phủ, ta thường dựa vào độ cao của địa hình. Nhưng đây là địa hình phẳng, nên không thể dựa vào độ cao được, do đó ta sẽ dùng một texture làm map để xác định loại địa hình sẽ phủ.

**Hình xxx-3-1** Một ví dụ về texture map

Như ta thấy ở hình trên, màu đen biểu thị chỗ có cỏ chẳng hạn, màu trắng biểu thị chỗ có đất, còn màu xám sẽ là sự hòa trộn giữa cỏ và đất.

Việc xác định loại texture để vẽ được thực hiện trong fragment shader:

|  |
| --- |
| struct TerrainProperties  {  vec3 TileRepeatFactor;  vec2 Threshold;  };  varying vec2 v\_texC;  uniform sampler2D u\_diffuseSampler1;  uniform sampler2D u\_diffuseSampler2;  uniform sampler2D u\_blendSampler;  uniform TerrainProperties u\_properties;  void main()  {  vec2 texC = vec2(v\_texC.x, 1.0 - v\_texC.y);  vec4 color1 = texture2D(  u\_diffuseSampler1,  texC \* u\_properties.TileRepeatFactor.x );  vec4 color2 = texture2D(  u\_diffuseSampler2,  texC \* u\_properties.TileRepeatFactor.y );  vec4 blend = texture2D(  u\_blendSampler,  texC \* u\_properties.TileRepeatFactor.z );    gl\_FragColor = mix(color1, color2, blend.x);  } |

1. **Chiếu sáng cảnh vật**

Một game mà không có ánh sáng sẽ trông vô cùng đơn điệu. Ánh sáng trong game không chỉ làm sinh động cảnh vật mà còn giúp ta nhìn rõ các đối tượng hơn.

* **Vấn đề 1**: Nguyên lý về chiếu sáng

Có nhiều cách chiếu sáng cảnh vật, nhưng ở đây ta chỉ khảo sát cách phổ biến nhất là mô hình chiếu sáng Phong (Phong reflection model).

* **Giải pháp**:

Trong mô hình Phong, có 3 thành phần tham gia vào cường độ sáng của một đối tượng:

* Ánh sáng lân cận (Ambient)
* Ánh sáng khuếch tán (Diffuse)
* Ánh sáng phản xạ (Specular)

**Hình xxx-4-1** Các thành phần ánh sáng trong mô hình Phong

Ánh sáng lân cận là ánh sáng mà vật trong bóng tối nhận được do vật được chiếu sáng phản xạ lại. Giá trí ánh sáng này giống nhau cho mọi vật trong tối:

|  |
| --- |
| [Toán tử là phép nhân từng thành phần của 2 vector. Ví dụ: .]  Trong đó: là tổng lượng ánh sáng lân cận vật nhận được  là khả năng hấp thụ và phản xạ ánh sáng lân cận của vật |

Ánh sáng khuếch tán là ánh sáng trực tiếp từ nguồn sáng mà vật nhận được:

|  |
| --- |
| Trong đó: là màu ánh sáng trực tiếp  là khả năng hấp thụ và phản xạ ánh sáng trực tiếp  là chiều của ánh sáng  là vector normal của vật |

Ánh sáng phản xạ là ánh sáng mà mắt nhận được do sự phản xạ từ vật bị chiếu sáng:

**Hình xxx-4-2** Mô hình ánh sáng phản xạ

|  |
| --- |
| Trong đó: là màu của ánh sáng phản xạ ra  là tổng lượng ánh sáng mà vật phản xạ ra |

Cường độ ánh sáng mà vật nhận được theo mô hình Phong là tổng của 3 thành phần trên:

|  |
| --- |
|  |

|  |
| --- |
|  |

* **Vấn đề 2**: Cài đặt chiếu sáng cảnh vật trong game
* **Giải pháp**:

Để cài đặt ánh sáng trong game, ta có 2 cách:

* Cài đặt trong vertex shader (Gouraud shading): Chạy nhanh hơn nhưng ra kết quả xấu hơn
* Cài đặt trong fragment shader (Phong shading): Chạy chậm hơn nhưng đẹp hơn

Nhóm em chọn cài đặt trong fragment shader:

|  |
| --- |
| struct MATERIAL  {  vec3 Ambient;  vec4 Diffuse;  vec4 Specular;  float Shininess;  };  struct LIGHT  {  // w == 0.0f: Directional light  // w == 1.0f: Point light  vec4 PosOrDir;  };  varying vec3 v\_norW;  varying vec2 v\_texC;  varying vec3 v\_surfaceToEye;  uniform sampler2D u\_diffuseSampler;  uniform MATERIAL u\_material;  uniform LIGHT u\_light;  void main() // Fragment shader  {  vec3 norW = normalize(v\_norW);  vec3 lightDir = -normalize(u\_light.PosOrDir.xyz);  vec3 surfaceToEye = normalize( v\_surfaceToEye );  // Get texture color  vec4 diffTexture = texture2D( u\_diffuseSampler, v\_texC );    // Apply ambient color  gl\_FragColor.rgb = u\_material.Ambient;  gl\_FragColor.a = 0.0;    // Apply diffuse color  float diffFactor = max( 0.0, dot( norW, lightDir ) );  gl\_FragColor.rgba += diffTexture \* u\_material.Diffuse \* diffFactor;    // Apply specular color  vec3 halfVector = normalize( lightDir + surfaceToEye );  float specFactor = pow(  max( 0.0, dot( norW, halfVector ) ),  u\_material.Shininess );  gl\_FragColor.rgba += u\_material.Specular \* specFactor;  } |

1. **Di chuyển nhân vật trong game**

Mô hình 3D chỉ là cái xác của nhân vật, việc chúng ta cho nhân vật di chuyển trong môi trường game mới là thổi hồn vào nhân vật.

* **Vấn đề 1**: Cho nhân vật di chuyển trong game
* **Giải pháp**:

Ta sẽ xây dựng lớp cơ sở MovingEntity quản lí việc di chuyển của nhân vật, mọi đối tượng di chuyển trong game đều kế thừa lớp này:

|  |
| --- |
| **class** MovingEntity  {  **public**:  // rotYOffset: When rotY = 0, angle between entity's heading and Oz  // -180 <= rotYOffset <= 180  // counter-clockwise (toward Oz) is positive direction  **virtual** **void** init(  **const** MyVec3& pos,  **const** MyVec3& target,  **const** MyVec3& rot,  **float** rotYOffset,  **float** speed,  **float** turnSpeed);  **virtual** **void** init(  **const** std::vector<MyVec3>& path,  **float** rotYOffset,  **float** speed,  **float** turnSpeed);  **virtual** **void** update(Timer& timer);  **void** turnTo(**const** MyVec3& target);  // Getter  MyVec3 getPos();  MyVec3 getRot();  **bool** isMoving();  // Setter  **void** accelerate(**float** dSpeed);  **void** disFollowPath();  **void** reFollowPath();  **protected**:  MyVec3 m\_position;  MyVec3 m\_target;  MyVec3 m\_rotation;  **float** m\_rotYOffset;  **float** m\_currOrientation;    **float** m\_speed;  **float** m\_turnSpeed;  **bool** m\_isMoving;  std::vector<MyVec3> m\_path;  **int** m\_pathPivot;  **int** m\_deltaPathPivot;  **bool** m\_followingPath;  }; |

* **Vấn đề 2**: Cho nhân vật xoay về hướng đang đi

Ta sẽ không bắt nhân vật ngay lập tức xoay về hướng đang đi, mà nhân vật sẽ từ từ xoay về hướng đang đi. Điều này sẽ tạo ra hiệu ứng di chuyển chân thật hơn.

* **Giải pháp**:

**Hình xxx-5-1** Mô hình di chuyển của nhân vật

Như ta thấy ở hình trên, nhân vật không ngay lập tức xoay về mục tiêu, mà chỉ một góc nhỏ hơn. Sau đó nhân vật sẽ thay đổi vị trí ứng với góc xoay, do đó nhân vật đi theo một đường vòng thay vì đi thẳng đến mục tiêu.

|  |
| --- |
| **static** **float** turnToFace(  **float** elapsedTime,  MyVec3 position,  MyVec3 faceThis,  **float** currentAngle,  **float** turnSpeed)  {  **float** x = faceThis.x - position.x;  **float** z = faceThis.z - position.z;  **float** desiredAngle = dATan2(x, z);    **float** difference = wrapAngle(desiredAngle - currentAngle);  difference = clamp(  difference,  -turnSpeed \* elapsedTime,  turnSpeed \* elapsedTime);  **return** wrapAngle(currentAngle + difference);  }  ...  m\_currOrientation = turnToFace(  timer.getElapsedTime(),  m\_position,  m\_target,  m\_currOrientation,  m\_turnSpeed);  MyVec3 heading = m\_speed \* MyVec3(  dSin\_optimized(m\_currOrientation), 0,  dCos\_optimized(m\_currOrientation));  m\_position += heading \* timer.getElapsedTime(); |

* **Vấn đề 3**: Một vài kĩ thuật AI
* **Giải pháp**:

Để nhân vật có thể tự động đi trên địa hình, ta dùng kĩ thuật path-following:

|  |
| --- |
| **if** (!m\_path.empty() && m\_followingPath)  {  m\_pathPivot += m\_deltaPathPivot;  **if** ((m\_pathPivot < m\_path.size()) && (m\_pathPivot >= 0))  {  setTarget(m\_path[m\_pathPivot]);  }  } |

Khi nhân vật đuổi theo kẻ thù, nếu khoảng cách lớn hơn một hằng số thì nhân vật sẽ không đuổi theo nữa:

|  |
| --- |
| **if** (obj->m\_atkTarget != **nullptr**)  {  **if** (  distance\_optimized(  obj->getPos(),  obj->m\_atkTarget->getPos()) > obj->m\_chasingRange  )  {  obj->m\_movingEnt.reFollowPath();  obj->m\_stateMachine->ChangeState(PawnState\_Walk::instance());  }  } |

1. **Xây dựng giao diện người dùng**

Giao diện người dùng là các đối tượng 2D trên màn hình như button, label, list, …

* **Vấn đề**: Xây dựng một thư viện cho phép các đối tượng 2D lên màn hình
* **Giải pháp**:

Xây dựng một lớp chung cho tấc cả các đối tượng giao diện kế thừa:

|  |
| --- |
| **class** UIWidget : **public** OnPressListenee  {  **public**:  **enum** Status  {  ACTIVE,  INACTIVE,  HIDDEN,  };  **public**:  UIWidget();  **virtual** ~UIWidget();  **virtual** **void** init(  **const** MyString& id,  **const** MyVec2& pos,  **const** MyVec2& size);  **virtual** **void** update(UserInput& userInput);  **virtual** **void** render(  SpriteBatch& spriteBatch,  **const** Rect2D\* viewport = **nullptr**) = 0;  // Getter  MyString getId()**const**;  **const** MyVec2& getSize()**const**;  **const** MyVec2& getPos()**const**;  // Setter  **void** setPos(**const** MyVec2& pos);  **void** setStatus(Status status);  **protected**:  MyString m\_id;  Rect2D m\_bounding;  Status m\_status;  }; |

Lớp cơ sở này sẽ chứa tấc cả các thuộc tính chung và làm nhiệm vụ kiểm tra xem đối tượng giao diện có được người dùng nhấn hay không.

Các đối tượng giao diện sẽ kế thừa lớp cơ sở trên và bổ sung các tính năng cho phù hợp.

1. **Âm thanh trong game**

Âm thanh là một phần không thể thiếu trong game, có thể nói game không có âm thanh giống như mất đi 50% sức hấp dẫn người chơi.

* **Vấn đề 1**: Chọn thư viện âm thanh
* **Giải pháp**:

Trong Android có sẵn thư viện âm thanh OpenSL ES, đây là một thư viện đa nền tảng, hỗ trợ nhiều định dạng như mpeg, wav, ogg, … Do đó nhóm chúng em sẽ sử dụng thư viện này.

* **Vấn đề 2**: Đưa âm thanh vào trong game
* **Giải pháp**:

Vì cách sử dụng OpenSL ES khá phức tạp, nên sẽ bọc lại thư viện này với một giao diện dễ sử dụng hơn.

Lớp AudioSystem có nhiệm vụ khởi tạo hệ thống:

|  |
| --- |
| **class** AudioSystem  {  **public**:  AudioSystem();  ~AudioSystem();  **private**:  SLresult initSLES();  **void** destroySLES();  **private**:  SLObjectItf m\_engineObject;  SLEngineItf m\_engine;  SLObjectItf m\_outputMixObject;  **private**:  **friend** **class** Audio;  }; |

Lớp Audio đại diện cho một tập tin âm thanh:

|  |
| --- |
| **class** Audio  {  **public**:  Audio();  ~Audio();  SLresult init(**const** MyString& filename, **bool** loop = **false**);  **void** play();  **void** stop();  **void** setVolume(**float** volume);  **private**:  **void** destroy();  **private**:  SLObjectItf m\_playerObject;  SLPlayItf m\_playerPlay;  SLVolumeItf m\_volumeItf;  SLSeekItf m\_seekItf;  **private**:  **static** AudioSystem m\_audioSys;  }; |