Đề tài: Lập trình game Metroid

**Lời mở đầu**

**Mục lục**

I. Giới thiệu về đề tài:

1. Tổng quan về đề tài:

Lập trình game là đã và đang là lĩnh vực đươc quan tâm và chú trọng hàng đầu trong ngành công nghiệp nội dung số. Với doanh số toàn cầu tăng trưởng chóng mặt, lập trình game đã trở thành ngành có tốc độ phát triển nhanh nhất trong lĩnh vực giải trí.

Ở Việt Nam, nghề lập trình game tuy đã xuất hiện từ rất lâu tuy nhiên chỉ ở những năm gần đây khi mà thị trường smartphone bùng nổ và đặc biệt là sự thành công của Nguyễn Hà Đông với tựa game Flappy Bird, nghề lập trình game được quan tâm hơn bao giờ hết.

Game Metroid: ………..

**(Ảnh game)**

2. Công nghệ sử dụng:

Visual Studio: ……..

DirectX9: ………..

Tiled map editor:

RapidXML:

3. Mục tiêu:

………………

II. Thiết kế:

1. Framework:

Trước khi đi vào phân tích cụ thể game Metroid, việc đầu tiên chúng ta cần phải biết là các thành phần căn bản trong game và làm sao để vẽ các nhân vật chúng ta muốn lên màn hình, cách game hoạt động và nhiều thứ khác. Để làm các công việc này trở nên có dễ dàng và có tính hệ thống hơn, ta cần một framework.

Vậy framework là gì?

…………………………….

Các lớp đối tượng chính có trong framework sẽ được phân tích một cách rõ ràng dưới đây.

1.1. Class Game:

Lớp Game thực hiện việc khởi tạo và cài đặt các thành phần cần thiết cho game như cửa sổ Game, directX, Input. Việc thực hiện cài đặt cho cửa sổ Game, directX hay Input đã được giải thích rất rõ ràng tại sách tham khảo.

**Nguồn sách: ………**

Ngoài ra lớp Game còn thực hiện việc quản lý các Scene và sự chuyển đổi Scene này qua Scene khác.

Các hàm căn bản nhất chúng ta cần biết ở lớp này là:

* **void virtual CreateGame() = 0:** Đây là hàm thuần ảo và hàm này được gọi 1 lần duy nhất ngay khi game vừa bắt đầu. Mục đích của việc sử dụng hàm thuần ảo là nhắc nhở cho người dùng nhớ việc override lại hàm này. Nếu hàm này không được override, lớp Game sẽ là lớp thuần ảo và không thể được khởi tạo.
* **void virtual UpdateGame(float dt) = 0:** Được gọi sau mỗi frame (Các frame thường được gọi sau mỗi 1/60s). Mọi hoạt động của game như vẽ các texture, sprite, handle input đều xảy ra ở hàm này.
* **void virtual Release() = 0:** Giải phóng các tài nguyên cần thiết để tránh memory leak như texture, sound, … Việc gọi hàm này cần được thực hiện ngay sau khi game kết thúc.
* **void setScene(Scene\* scene):** Xác định Scene hiện tại sẽ được render.

1.2. Class Scene:

Chúng ta hoàn toàn có thể viết toàn bộ nội dung game vào class Game ở trên vì dường như nó đã cung cấp đủ các phương thức cần thiết để cho một game có thể hoạt động bình thường. Tuy nhiên, việc viết tất cả nội dung game vào chỉ một class Game rất khó để quản lý nếu chúng ta có khá nhiều cảnh game. Ví dụ MenuScene, PlayScene, GameOverScene, …

Vì vậy, Framework cung cấp sẵn lớp Scene để việc quản lý các cảnh game dễ dàng hơn.

Các hàm căn bản nhất của lớp Scene là:

* **void virtual Create()=0:** Khởi tạo các tài nguyên và lớp cần thiết.
* **void virtual Update(float dt) = 0:** Tương tự như lớp Game, hàm này được gọi mỗi frame và mọi hoạt động của game được diễn ra tại đây.
* **void virtual Release() = 0:** Giải phóng các tài nguyên cần thiết để tránh memory leak.

Các hàm căn bản của của lớp Scene gần như không có gì quá khác biệt với các hàm căn bản của lớp Game. Điều này là dễ hiểu vì thực chất, lớp Game thực hiện việc update cho lớp Scene ở hàm **UpdateGame(float dt)**. Cách lớp Game thực hiện update cho Scene sẽ được nói rõ hơn khi ta đi vào phân tích Game Metroid cụ thể.

Phải lưu ý rằng, việc thực hiên giải phóng Scene (gọi hàm Release()) không được gọi tự động mà chúng ta phải tự quản lý lúc nào thì Scene được giải phóng, lúc nào thì không. Lý do là khi trong game, một Scene như MenuScene có thể được sử dụng lại nhiều lần và một số Scene khác có thể chỉ cần sử dụng một lần.

1.3. Class Texture:

Lớp Texture đơn giản là lớp dùng để giải mã một bức ảnh từ đường dẫn nào đó để về sau nó có thể vẽ lên màn hình. Nói cách khác, chúng ta có thể gọi Texture là một bức ảnh.

Các hàm căn bản nhất của lớp này là:

* **Texture(const std::string &filePath):** constructor để khởi tạo texture với một file hình ảnh đã có.

Đây là phương thức duy nhất mà chúng ta cần biết để chuẩn bị cho việc vẽ thứ gì đó hay ho lên màn hình. Đương nhiên, nếu cần phải lấy một vài thông số khác của bức ảnh chúng ta đã load lên như chiều dài, chiều rộng của bức ảnh thì Framework cũng có cung cấp một vài hàm khác như getImageSize().



Figure 1 Trục và hình ảnh Texture được vẽ lên màn hình

1.4. Class TextureRegion: **(Cần sửa - Vinh)**

Hầu hết thời gian trong game, chúng ta có rất nhiều hình ảnh để vẽ. Giả sử để vẽ được 100 nhân vật lên màn hình, có vẻ như chúng ta cần 100 Texture. Đây là điều chính xác theo suy nghĩ thông thường tuy nhiên việc giải mã 100 hình ảnh này và load lên Texture là một công đoạn khá “đắt đỏ” cho GPU bởi vì các hình ảnh này thực chất được load lên GPU sẵn trước khi được vẽ lên màn hình.

TextureRegion giải quyết điều này thông qua việc chúng ta chỉ cần load một bức ảnh duy nhất chứa 100 nhân vật này và TextureRegion cho phép chúng ta vẽ một phần trong bức ảnh đó.



Figure 2 SpriteSheet và tọa độ các nhân vật trong SpriteSheet



Figure 3 Hình ảnh khi vẽ lên màn hình

Các hàm căn bản của TextureRegion là:

* [**TextureRegion**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/TextureRegion.html#TextureRegion-com.badlogic.gdx.graphics.Texture-float-float-float-float-)**(**[**Texture**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/Texture.html)**texture, float rectLeft, float rectTop, float rectWidth, float rectHeight):** Constructor với một texture và một vùng cụ thể. Ví dụ trong Figure 2, để vẽ được góc phần tư thứ nhất của SpriteSheet, x, y, width, height sẽ là: (x,y) = (0,0) là góc phía trên bên trái và (width, height) = (64,64) là chiều dài và chiều rộng của vùng cần vẽ.
* **void SetRectPosition(float rectX, float rectY):** hàm set ví trí rect tương tự như ở constructor
* **void SetRectSize(float rectWidth, float rectHeight):** hàm set kích thước rect tương tự như ở constructor
* **setTexture(Texture texture):** set texture cho TextureRegion này.
* **…**

Có một điều phải để ý là lớp TextureRegion và một số lớp liên quan đến việc vẽ hình khác không có phương thức dispose(). Tại sao lại như vậy?

Lý do là lớp TextureRegion chỉ thực hiện việc lưu trữ các thông số là các kiểu dữ liệu cơ sở như kiểu int. TextureRegion có lưu trữ kiểu Texture tuy nhiên việc lưu trữ này chỉ là lưu trữ dưới dạng reference. Vì vậy, khi chúng ta ghi nhớ giải phóng cho lớp Texture mà chúng ta đã truyền vào cho lớp TextureRegion này thì phương thức dispose() cho lớp này là không cần thiết.

2.1.5. Class Animation: **(Cần sửa - Vinh)**

Trước khi đi vào phân tích lớp Animation, chúng ta cần phải hiểu Animation là gì. Animation thực chất là việc hoạt họa nhân vật bằng cách cho các bức ảnh của nhân vật có liên quan đến một chuyển động nào đó chạy liên tiếp.

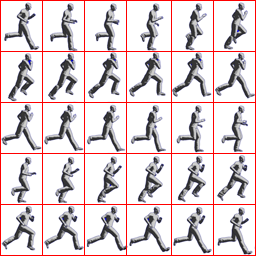


Figure 4 Ví dụ về Animation

Bằng cách cho các bức ảnh này chạy liên tiếp và tuần tự từ trái sang phải và từ trên xuống dưới ta sẽ có được hoạt cảnh của nhân vật.

Lớp Animation thực hiện lưu trữ một danh sách các đối tượng làm đại diện cho việc hoạt họa nhân vật như thế này. Lớp Animation là một template class dạng: Animation<T>. Nghĩa là, lớp Animation có thể chứa một danh sách các Texture, TextureRegion hoặc Sprite (hoặc một số lớp khác tương tự). Lớp Animation<TextureRegion> là lớp mà chúng ta thường sử dụng nhất vì tính tiện dụng của nó như đã phân tích ở lớp Sprite.

Các hàm chính của lớp Animation là:

* **Animation(float frameDuration, Array<T> keyFrames):** Constructor với frameDuration là độ dài của một frame trong Animation (nghĩa là thời gian giữa việc chuyển từ một hoạt cảnh này sang một hoạt cảnh khác) và mảng các hoạt cảnh gọi là keyFrames.
* **getKeyFrame(float stateTime):** trả về một hoạt cảnh dựa trên thời gian trạng thái (stateTime).
* …

2.1.6. Class SpriteBatch: **(Cần sửa – Vinh hoặc Sơn)**

Có vẻ chúng ta quên gì đó? Có rất nhiều lớp đối tượng liên quan đến hình vẽ mà chúng ta đã nói ở trên. Tuy nhiên, làm sao để vẽ chúng?

Việc vẽ tất các các lớp đối tượng trên được thực hiện bởi lớp SpriteBatch. Các hàm chính của lớp này là:

* **SpriteBatch():** constructor.
* **begin():** bắt đầu việc vẽ.
* **end():** kết thúc việc vẽ.
* [**draw**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/SpriteBatch.html#draw-com.badlogic.gdx.graphics.Texture-float-float-)**(**[**Texture**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/Texture.html)**texture, float x, float y):** vẽ texture tại vị trí (x,y)
* [**draw**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/SpriteBatch.html#draw-com.badlogic.gdx.graphics.Texture-float-float-float-float-)**(**[**Texture**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/Texture.html)**texture, float x, float y, float width, float height):** vẽ texture tại vị trí (x,y) và với kích thước (width, height)
* [**draw**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/SpriteBatch.html#draw-com.badlogic.gdx.graphics.g2d.TextureRegion-float-float-)**(**[**TextureRegion**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/TextureRegion.html)**region, float x, float y):** vẽ TextureRegion tại vị trí (x,y)
* **dispose():** giải phóng tài nguyên.

Lại một câu hỏi khác là tại sao không cung cấp cho mỗi lớp một phương thức vẽ riêng mà phải thông qua một lớp khác là SpriteBatch? Tính toán vị trí hình học thích hợp cho việc vẽ một bức hình là một việc khá “đắt đỏ”. Vì vậy, tất cả việc vẽ phải được thực hiện giữa begin() và end() để giảm thiểu chi phí này.

(Có thể đọc thêm về các lớp để thực hiện vẽ đối tượng tại đây: <https://github.com/libgdx/libgdx/wiki/Spritebatch%2C-Textureregions%2C-and-Sprites>)

2.1.7: Class Sprite: **(Cần sửa - Vinh)**

Class Sprite thực chất là một lớp kế thừa là lớp TextureRegion tuy nhiên có thêm một số tính năng như set kích cỡ, màu, góc xoay, …

Các hàm căn bản của lớp Sprite là:

* **Sprite(Texture texture):** Constructor với một texture.
* **Sprite(Texture texture, int srcX, int srcY, int srcWidth, int srcHeight):** Constructor với một texture và một vùng cụ thể tương tự như đối với TextureRegion.
* **Sprite(TextureRegion region):** Constructor với một một vùng cụ thể.
* [**setBounds**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/Sprite.html#setBounds-float-float-float-float-)**(float x, float y, float width, float height):** set vị trí và kích cỡ của sprite khi được vẽ lên màn hình.
* [**setColor**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/Sprite.html#setColor-float-float-float-float-)**(float r, float g, float b, float a):** set màu cho sprite.
* [**setPosition**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/Sprite.html#setPosition-float-float-)**(float x, float y):** set vị trí.
* [**setSize**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/Sprite.html#setSize-float-float-)**(float width, float height):** set kích cỡ.
* **setRegion(TextureRegion region):** set region cho sprite.
* **Draw(Batch batch):** vẽ Sprite này với các thông số đã set cho lớp Sprite này. Lưu ý hàm này phải được gọi giữa batch.begin() và batch.end().

Câu hỏi đặt ra ở đây là tại sao LibGDX lại mất thời gian tạo ra một lớp Sprite như thế này thay vì bổ sung các hàm tương tự cho lớp TextureRegion. Câu trả lời là vì tính tiện dụng của nó. Hầu hết các đối tượng trong thế giới game của chúng ta sẽ được kế thừa lại một lớp hình ảnh nào đó cho sự tiện dụng trong việc vẽ lên màn hình cũng như thay đổi kích cỡ hay một vài thứ khác. Giả sử ta có lớp Player là một lớp được kế thừa từ lớp TextureRegion và giả sử lớp TextureRegion có đầy đủ các tính năng như là lớp Sprite. Trong game, việc Player thay đổi từ trạng thái này qua trạng thái khác sẽ dẫn đến việc các hình ảnh hiển thị sẽ thay đổi và có lẽ dẫn đến cả việc thay đổi kích cỡ hình ảnh hiển thị của nhân vật. Lúc đó, chúng ta sẽ liên tục phải gọi lại hàm setRegion(int x, int y, int width, int height) và setTexture(Texture texture) của lớp TextureRegion. Điều này tuy không khó nhưng khá tốn thời gian và sự gọn gàng của code. Vì vậy, LibGDX cung cấp cho ta lớp Sprite và đơn giản hóa lớp TextureRegion để làm những công việc tương tự như thế này. Tất cả việc chúng ta phải làm là khởi tạo các TextureRegion cần thiết và gọi hàm **setRegion(TextureRegion region)** của lớp Sprite. Việc này sẽ gọn gàng hơn rất nhiều.

2.1.8. Class Camera:

Thế giới game rất rộng lớn và chúng ta không nhất thiết phải nhìn toàn bộ nó trong hầu hết tựa game. Camera đóng vai trò như một cửa sổ khi chúng ta nhìn vào thế giới, chúng ta chỉ cần nhìn vào một phần trong game và thay đổi góc nhìn khi cần thiết.

Đối với các framework cho phép lập trình các game như 2.5D hay 3D, Lớp Camera có thể rất phức tạp. Tuy nhiên đối với Framework 2D hiện tại trong nội dung bài viết này, nó chỉ đơn giản là một điểm trong thế giới game và là trung điểm của “cửa sổ”.

2 phương thức chính của lớp Camera là:

* **const Vector2& GetPosition()**: lấy vị trí hiện tại của camera.
* **void SetPosition(float x, float y)**: set vị trí cho camera.

2.1.9. Class Viewport: **(Chưa có trong framework. Có thể bổ sung sau)**

2.1.10. Class Body: **(Thiện)**

2.1.11: Class World: **(Trí hoặc Thiện)**

Thế giới game, là nơi chứa tất cả các vật thể, thuộc tính của game, được mô phỏng dựa trên thế giới thực tại ở một mức độ nào đó. Do nguyên lí hoạt động của một game hiện nay là ghi lại trạng thái của tất cả các vật thể tại một thời điểm nhất định rồi thể hiện lại trạng thái của chúng thông qua một “khung cửa sổ” có giới hạn có thể nhìn vào bên trong thế giới ảo – là camera, nên thế giới game không đơn thuần chỉ là một định nghĩa dùng để chứa tất cả cả thực thể của game, mà còn có thể đóng vai trò quản lí việc cập nhật trạng thái của các thực thể đó.

Đối với framework dành cho lập trình game 2D, lớp World có thể được định nghĩa là một đối tượng chứa Camera – là một điểm trong thế giới game và cũng là trung điểm của “cửa sổ” dành cho việc thể hiện các đối tượng trong game ra màn hình. Bên cạnh đó, lớp World cũng bao gồm tất cả các đối tượng trong game và mối liên hệ giữa chúng với nhau, ngoài ra còn có chứa các thuộc tính thực tế như trọng lực…

Các phương thức chính của lớp World:

* **SetCamera(Camera \*cam):** Dùng để set một Camera được khởi tạo sẵn vào World.
* **SetGravity(float gravity):** Dùng để set trọng lực cho World, giá trị gravity ở đây là giá trị mong muốn cho vector Oy hướng xuống mặt đất.
* **Update(float dt):** Dùng để cập nhật trạng cho tất các các đối tượng có trong World sau một khoảng thời gian dt (khoảng thời gian cách nhau giữa 2 khoảnh khắc của World được thể hiện trên cửa sổ nhất) định.
* **Release():** Dùng để hủy bỏ tất cả các đối tượng trong World.

Đối với các phương thức **SetCamera, SetGravity, Release,** chức năng của chúng tương đối rõ ràng, còn đối với phương thức **Update**, chúng ta cần đi sâu hơn vào một chút.

Phương thức **Update** bao gồm các việc: xét va chạm giữa các đối tượng, giải quyết các vấn đề liên quan đến va chạm, cập nhật trạng thái (vị trí, vận tốc, …) của đối tượng sao cho phù hợp với logic của game đã được định trước.

2.1.12: Class BodyDef: **(Thiện)**

2.1.13: Class Collision **(Thiện)**

Các vật thể trong thế giới game trong khi tương tác với nhau sẽ dẫn đến việc chúng va chạm nhau. Và việc phân tích và xử lý các va chạm đó sẽ do Class Collision thực hiện. Về cốt lõi của class này, nhóm đã dựa trên thuật toán xử lý va chạm SweptAABB và thêm một số thay đổi để cho phù hợp với cấu trúc của framework.

Các phương thức chính của lớp Collision:

* **IsColliding(Body \*targetBody, Body \*otherBody, float DeltaTime)**: kiểm tra xem body của 2 vật thể có chuẩn bị va chạm với nhau không dựa trên thuật toán xử lý va chạm SweptAABB.
* **IsTouching(Body \*targetBody, Body \*otherBody)**: kiểm tra xem body của 2 vật thể có đang đứng sát kế bên nhau hay không.
* **IsOverlaying(Body \*targetBody, Body \*otherBody)**: kiểm tra xem body của 2 vật thể có đang nằm chồng lấp lên nhau hay không.
* **IsPreviousTouching(Body \*targetBody, Body \*otherBody)**: kiểm tra xem body 2 vật thể có đang đứng sát kế bên nhau trong lần kiểm tra va chạm trước hay không.
* **IsPreviousOverlayed(Body \*targetBody, Body \*otherBody)**: kiểm tra xem body của 2 vật thể có đang nằm chồng lấp lên nhau trong lần kiểm tra va chạm trước hay không.
* **GetBroadphaseRect(Body \*body, float DeltaTime)**: trả về 1 hình chữ nhật thể hiện sự bao phủ của khi di chuyển trong 1 khoảng thời gian DeltaTime của body 1 vật thể.
* **PerformCollision(Body \*targetBody, Body \*otherBody, float DeltaTime, int collisionAction, bool &needMoveX, bool &needMoveY)**: thực hiện việc quyết định xử lý đối với body của 2 vật thể dựa trên các kết quả thực hiện của các hàm trên.

Việc xét va chạm bằng lớp Collision sẽ được thực hiện trong quá trình update lớp World. Tại đó, lần lượt mỗi cặp 2 vật thể sẽ được kiểm tra va chạm.

2.1.14: Class Font: **(Trí)**

Lớp Font là một lớp phục vụ cho việc thiết kế giao diện trong game. Cụ thể, lớp Font ở đây được sử dụng để chứa kiểu chữ của hệ thống được sử dụng cho việc lập trình việc hiển thị số máu hiện tại của nhân vật thông qua một con số trên màn hình.

Các phương thức chính:

* **Font(const std::string &fontname):** phương thức khởi tạo với tên font phù hợp.
* **Release():** giải phóng đối tượng font.

2.1.15: Class Label: **(Trí)**

Lớp Label là một lớp phục vụ cho việc thiết kế giao diện trong game. Cụ thể, lớp Label ở đây được sử dụng để chứa và vẽ chuỗi hiển thị số máu hiện tại của nhân vật chính ra màn hình.

Các phương thức chính:

* **Draw(Camera \*cam):** Vẽ Label tại vị trí dựa trên vị trí của camera.
* **SetPosition(float x, float y):** Đặt lại vị trí mong muốn cho Label.
* **SetSize(float width, float height):** Đặt lại kích thước mong muốn cho Label.
* **SetFont(Font \*font):** Đặt lại Font mong muốn cho Label.

Tại mỗi frame, Label sẽ được vẽ dựa trên vị trí của camera do đó nó sẽ luôn có khoảng cách cố định so với vị trí của điểm Camera, tạo nên cảm giác cố định tại một vị trí trên cửa sổ, nhưng thật ra tọa độ của Label luôn thay đổi dựa trên tọa độ của camera trong world. Sau khi đã được vẽ xong, ta tiến hành đặt lại vị trí của Label (cập nhật) theo vị trí của camera sau một khoảng thời gian dt.

2.1.16: Class CollisionQuadTree **(Trí)**:

Lớp CollisionQuadTree là lớp chứa cấu trúc cây tứ phân dùng để phân hoạch các đối tượng trong game vào các nút trên cây tứ phân dựa trên tương quan về vị trí của chúng trong world bằng cách phân hoạch không gian thành 4 vùng riêng biệt và nhỏ dần ở những cấp thấp hơn theo quy tắc những đối tượng nào nằm trong cùng một vùng không gian với nhau thì sẽ thuộc chung một nút trên cây quadtree, giúp giảm tải cho việc thực hiện kiểm tra và giải quyết đụng độ giữa các đối tượng bằng cách bỏ qua những việc kiểm tra và xử lí va chạm giữa các đối tượng không thuộc cùng một nút trên quadtree – chắc chắn sẽ không va chạm tại thời điểm đang xét.

Các phương thức chính:

* **Split():** khi một nút trong quadtree có số lượng đối tượng vượt quá số lượng đối tượng tối đa được quy định sẵn thì phương thức Split sẽ tách nút đó ra thành 4 nút con và phân phối các đối tượng thuộc nút đó vào 4 nút con dựa trên thuật toán phân hoạch không gian.
* **Insert(Body\* body):** thêm một đối tượng vào nút phù hợp trên cây quadtree.
* **Clear():** xóa cây quadtree.
* **Update(Vector2 size, Vector2 position):** nhằm phục vụ cho thuật toán phân hoạch không gian, class CollisionQuadTree được thiết kế bao gồm các thuộc tính trong đó có 2 thuộc tính là vị trí của điểm trên cùng bên trái và kích thước của vùng không gian mà quadtree đó “quản lí”, vị trí khởi đầu và kích thước của vùng không gian của quadtree sẽ tương ứng với vị trí và kích thước của cửa sổ, cũng là vị trí của camera dịch lên nửa chiều dài và dịch trái nửa chiều rộng của sổ. Do vị trí camera luôn thay đổi nên ta phải thực hiện việc update 2 thuộc tính nói trên của quadtree.
* **Retrieve(std::vector<Body\*> &returnBodies, Body \*body):** lập và trả về một danh sách các đối tượng thuộc cùng một nút với đối tượng body.

Cách thức hoạt động của class CollisionQuadTree tương đối đơn giản và dễ hiểu. Trước khi bắt đầu việc cập nhật world, ta xóa bỏ tất cả các đối tượng trong quadtree, cập nhật lại thuộc tính vị trí và kích thước cho quadtree, sau đó dùng phương thức Insert(Body \*body) để thêm tất cả những đối tượng nằm trong vùng màn hình hiện tại vào cây quadtree, phương thức Insert đã được cài đặt theo thuật toán phân hoạch không gian và sẽ tự động thêm nút con cho cây quadtree nếu số lượng đối tượng của một nút vượt quá quy định. Sau khi đã có cây quadtree của màn hình ở frame hiện tại thì ta sẽ tiến hành xét và xử lí va chạm của từng đối tượng trong màn hình với danh sách các đối tượng có khả năng đụng độ với đối tượng đang xét nhờ phương thức Retrieve.

2.1.17: Class Input:

2.1.18: Class Sound **(Vinh)**

2.1.19: Các Class liên quan đến Map:

Việc xây dựng Map cho game không hề đơn giản và rất mất thời gian. Ở hầu hết các tựa game, người ta đều sử dụng một công cụ nào đó để giúp đỡ cho công việc này trở nên nhanh và đơn giản hơn.

Framework trong nội dung bài viết này cung cấp Class TMXMap hỗ trợ việc đọc file .tmx được tạo ra từ phần mềm Tiled Map Editor.

Tiled Map Editor là một công cụ mã nguồn mở được phát triển chủ yếu bởi Thorbjørn Lindeijer. Công cụ này cho phép người sử dụng tạo ra một map dạng tile một cách dễ dàng và nhanh chóng chỉ việc sử dụng các thao tác kéo thả đơn giản.

Đọc về công cụ này tại đây: <http://doc.mapeditor.org/en/latest/manual/introduction/>

2.1.19.1. Class TMXLoader:

Lớp TMXLoader cung cấp khả năng lưu trữ nhiều map cùng lúc.

Các phương thức chính của TMXLoader là:

* **void AddMap(const std::string& mapName, const std::string& filePath, float scale):** Load map từ filePath với tên mapName và scale map theo mong muốn (thường là 1 để giữ nguyên kích cỡ)
* **TMXMap\* GetMap(const std::string& mapName):** Trả về map đã load với tên map tương ứng.

Với phương thức **AddMap,** chúng ta có thể load nhiều map cùng lúc vào TMXLoader và sau đó sử dụng hàm **GetMap** để lấy ra map mà theo tên mà chúng ta đã đặt cho nó.

2.1.19.2. Class TMXTileLayer

Nếu đã đọc sơ qua về tài liệu của công cụ Tiled Map Editor, chúng ta sẽ biết TileLayer là gì. Một Tile Layer chính là phần hình ảnh của Map sau khi ta đã sử dụng các tile để vẽ map.

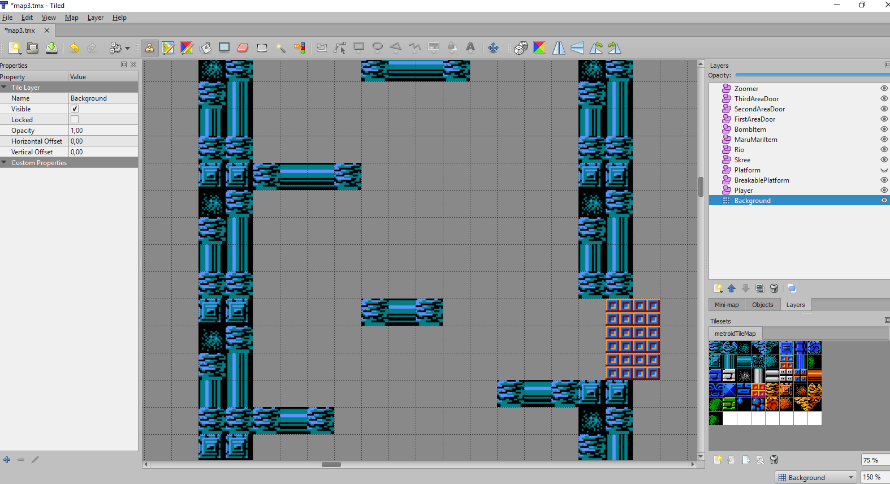


Figure 5 Ví dụ về Tile Layer

3 phương thức mà chúng ta cần quan tâm tại lớp này là:

* **unsigned int GetWidth():** trả về độ dài của layer.
* **unsigned int GetHeight():** trả về độ cao của layer.
* **unsigned int\*\* GetData():** trả về data của layer.

Framework trong nội dung bài còn khá hạn chế, vì vậy hiện tại Framework này chỉ cho phép đọc được data encoding CSV (dạng ma trận 2 chiều).

2.1.19.3. Class TMXObjectGroup

Tiled map editor cho phép đặt các Object vào map để thể hiện “cơ thể” của phần hình ảnh chúng ta đã tạo từ Tile Layer

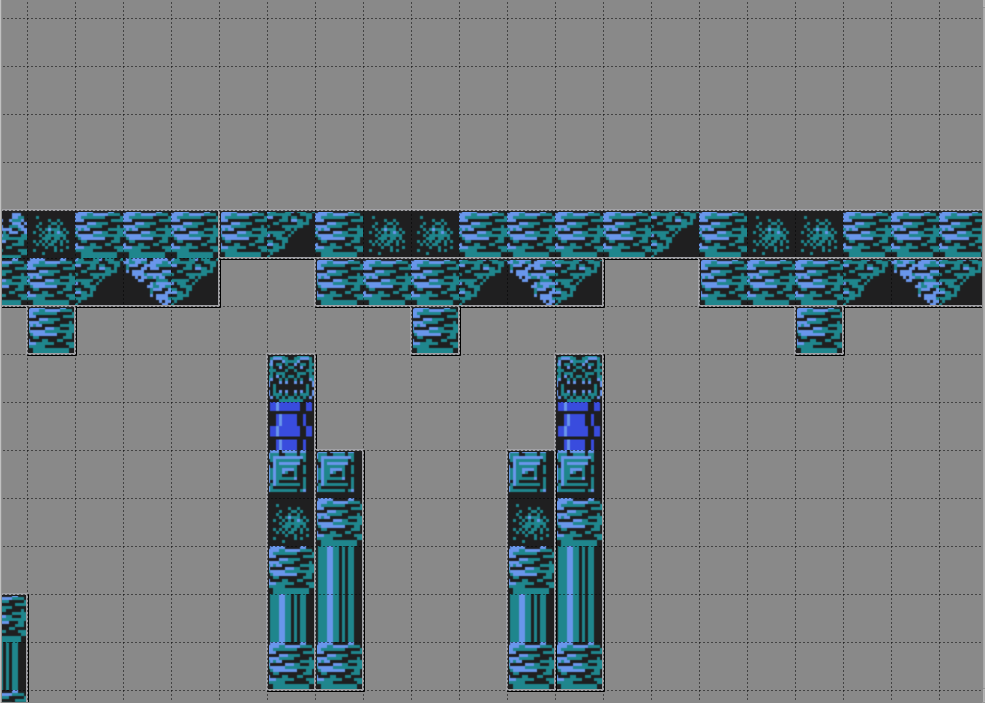


Figure 6 Ví dụ về ObjectGroup

Phương thức duy nhất mà chúng ta quan tâm về lớp này là:

* **const std::vector<Shape::Rectangle>& GetRects():** Lấy ra các Rectangle đại diện cho Object đã được vẽ.

2.1.19.4. Class TMXMap:

TMXMap chứa thông tin về tất các các tile layer và object group có trong map đó.

Các phương thức chính của lớp TMXMap là:

* **TMXObjectGroup\* GetObjectGroup(const std::string &groupName):** Trả về ObjectGroup theo tên đã đặt.
* **TMXTileLayer\* GetTileLayer(const std::string &tileLayerName):** Trả về tile layer theo tên đã đặt.
* **void Render(SpriteBatch \*batch):** Vẽ map này lên màn hình.

2. Hoạt động của Framework:

Game sẽ được chia các Scene và được quản lý bởi một lớp Game Manager (Kế thừa từ lớp Game) duy nhất. Để có thể hiểu một cách rõ ràng hơn về cách hoạt động của Framework, ta lấy ví dụ một số Scene cơ bản:

* Intro Scene
* Menu Scene
* Play Scene
* GameOver Scene

Cách thức game hoạt động được mô tả như hình bên dưới:

GameOver Scene

Các lệnh được gọi

Người chơi kết thúc game

Game được kết thúc

Bắt đầu game

Game Manager

Intro Scene

Menu Scene

Play Scene

Các Scene có thể chuyển đổi trực tiếp

Lớp Game quản lý việc thực hiện việc giữ cho game hoạt động và chuyển đổi giữa các Scene với nhau. Ví dụ, khi bắt đầu game, người dùng ở Intro Scene. Khi người chơi muốn chuyển sang Menu Scene, người dùng nhấn phím Enter. Lập tức, Intro Scene gửi một lệnh thông báo người dùng đã nhấn phím Enter và Game Manager thực hiện chuyển đổi sang Menu Scene.

Tại một Scene thông thường, quá trình Scene hoạt động thường được diễn ra như sau:

Người dùng chuyển đổi Scene

Scene tạm dừng hoặc kết thúc

Bắt đầu Scene

Xử lý vật lý

Cập nhật trạng thái nhân vật

Nhận Input

Render

Trên đây chỉ là một sơ đồ khuyên dùng cho hoạt động của một Scene. Đương nhiên, quá trình này có thể thay đổi hoặc thậm chí bỏ bớt một số bước như bước xử lý vật lý đối với các scene đơn giản như IntroScene hay MenuScene.

3. Game Metroid:

3.1. Intro Scene: **(Vinh)**

3.2. Menu Scene: **(Vinh)**

3.3. PlayScene: **(Ai siêng thì viết)**

3.3.1. Map:

3.3.2. Player:

3.3.3. Skree:

3.3.4. Zoomer:

3.3.5…………

3.4. GameOverScene: **(Chưa có code - Ai siêng thì code rồi viết luôn)**

III. Kết luận:

1. Kết luận:

2. Hướng mở rộng:

**Tài liệu:**