Đại HỌC QUỐC GIA TP.HỒ CHÍ MINH   
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN



BÁO CÁO ĐỒ ÁN MÔN HỌC  
NHẬP MÔN PHÁT TRIỂN GAME  
  
ĐỀ TÀI: LẬP TRÌNH GAME METROID

**GVHD: ĐINH NGUYỄN ANH DŨNG**

NHÓM SINH VIÊN THỰC HIỆN:

1. TRƯƠNG NGỌC SƠN - 15520740
2. TRẦN PHÚ VINH – 15521020
3. NGUYỄN ĐÌNH THIỆN – 15520831
4. TRẦN MINH TRÍ - 15520925

TP.HCM

01/01/2018

Mục lục

[**I. Giới thiệu về đề tài:** 2](#_Toc502606970)

[**1.** **Tổng quan về đề tài:** 2](#_Toc502606971)

[**2.** **Công nghệ sử dụng:** 2](#_Toc502606972)

[**II. Thiết kế:** 2](#_Toc502606973)

[**1.** **Framework:** 2](#_Toc502606974)

[1.1. Class Game: 2](#_Toc502606975)

[1.2. Class Scene: 3](#_Toc502606976)

[1.3. Class Texture: 3](#_Toc502606977)

[1.4. Class TextureRegion: 4](#_Toc502606978)

[1.5. Class Animation: 5](#_Toc502606979)

[1.6. Class SpriteBatch: 5](#_Toc502606980)

[1.7. Class Sprite: 6](#_Toc502606981)

[1.8. Class Camera: 7](#_Toc502606982)

[1.9. Class Body: 7](#_Toc502606983)

[1.10. Class World: 8](#_Toc502606984)

[1.11. Class Collision: 9](#_Toc502606985)

[1.12. Class Font: 10](#_Toc502606986)

[1.13. Class Label: 10](#_Toc502606987)

[1.14. Class CollisionQuadTree: 10](#_Toc502606988)

[1.15. Class Input: 11](#_Toc502606989)

[1.16. Class Sound 12](#_Toc502606990)

[1.17. Các Class liên quan đến Map: 12](#_Toc502606991)

[**2.** **Hoạt động của Framework:** 14](#_Toc502606992)

[**3.** **Game Metroid:** 16](#_Toc502606993)

[3.1. Intro Scene: 16](#_Toc502606994)

[3.2. Menu Scene: 16](#_Toc502606995)

[3.3. PlayScene: 16](#_Toc502606996)

[3.4. GameOverScene: 21](#_Toc502606997)

[**III. Kết luận:** 21](#_Toc502606998)

[**1.** **Kết quả đạt được:** 21](#_Toc502606999)

[**2.** **Hướng cải tiến và phát triển:** 21](#_Toc502607000)

[**Tài liệu tham khảo:** 22](#_Toc502607001)

[**Bảng phân công công việc** 22](#_Toc502607002)

# **Giới thiệu về đề tài:**

## **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.

## **Công nghệ sử dụng:**

* Visual Studio
* DirectX9
* Tiled map editor
* RapidXML

# **Thiết kế:**

## **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.

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

### 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: Giáo trình Nhập môn công nghệ phần mềm**

**(Tác giả: PGS. TS. Vũ Thanh Nguyên,**

**Ths. Đinh Nguyễn Anh Dũng)**

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.

### 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.

### 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().

### Class TextureRegion:

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.

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.

### Class Animation:

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.

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).

### Class SpriteBatch:

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>)

### Class Sprite:

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.

### 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.

### Class Body:

Mọi vật trong thế giới game trong game cần phải tương tác với nhau để cho người chơi có một trải nghiệm giống với thực tế. Và thành phần cần thiết để các vật thể có thể tương tác được với nhau đó là 1 “cơ thể”. Cơ thể của các vật thể trong game chỉ là 1 hình chữ nhật với cái thuộc tính cơ bản như:

* **Vector2 \_Position**: chỉ ra toạ độ x và y của body trong thế giới. Lưu ý rằng toạ độ này là toạ độ của điểm tâm chính giữa của hình chữ nhật.
* **Vector2 \_PreviousPosition**: chỉ ra toạ độ x và y của body trong frame trước đó trong thế giới. Lưu ý rằng toạ độ này là toạ độ của điểm tâm chính giữa của hình chữ nhật.
* **Vector2 \_Size**: chỉ ra chiều ngang và chiều cao của body, với x là chiều ngang và y là chiều cao.
* **Vector2 \_LinearDrag**: chỉ ra lực cản của gió dành cho body này, lực cản này được sử dụng để tính toán vật lý cho vật thể.
* **unsigned long categoryBits:** chỉ ra body là loại vật thể gì trong game. Trong game, mỗi loại quái vật, người chơi, mặt phẳng đều có 1 ký hiệu riêng để dễ phân biệt. Danh sách các bit dành cho các loại vật thể được khai báo trong file CollisionBit.h
* **unsigned long maskBits:** chỉ ra body này sẽ có thể tương tác với loại vật thể nào trong game. Ví dụ như Player có categoryBits là 1 và maskBits là 2. Mặt phẳng có categoryBits là 2 và maskBits là 1 thì Player có thể tương tác với mặt phẳng và ngược lại.
* **BodyType \_BodyType**: phân loại của body. Có tất cả 3 kiểu body trong framework. Đặc tính của mỗi kiểu được thể hiện trong bảng sau:

enum BodyType {

Static // nếu mang kiểu static thì body sẽ bất động, không tác động được // lực đẩy hay vận tốc nào, nhưng sẽ vẫn được kiểm tra va chạm.  
 // Các thể có kiểu body này thường là các mặt phẳng, hoặc là các   
 // quái vật khi bị đạn của player bắn.

, Kinematic // nếu mang kiểu kinematic thì body vẫn sẽ di chuyển bằng vận // tốc, nhưng sẽ không bị ảnh hưởng bởi lực hút của thể giới // game.

, Dynamic // nếu mang kiểu dynamic thì body sẽ hoạt động bình thường // tương tự như 1 đối tượng ngoài thực tế. Body sẽ có thể di // chuyển bằng vận tốc, bị ảnh hưởng bởi lực hút của thể giới // game

};

* **std::string \_ID**: số định danh của body
* **GameObject\* \_Extra**: 1 đối tượng trong game đi kèm với object. Đối tượng có thể là bất cứ gì
* **bool \_IsSensor**: là true nếu body được các body phụ trợ khác đi kèm, là false nếu không được phụ trợ bởi body nào khác. Ví dụ như player được phụ trợ thêm 2 body nhỏ khác là foot và head.

Các phương thức cơ bản: hầu hết là các phương thức get va set các thuộc tính trên.

### Class World:

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.

### Class Collision:

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.

### Class Font:

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.

### Class Label:

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.

### Class CollisionQuadTree:

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.

### Class Input:

Framework cung cấp lớp Input phục vụ cho việc nhận các thao tác phím từ bàn phím. Các phương thức chính của lớp này là:

* **bool GetKey(int keyCode)**: Trả về true trong khi một phím đang được giữ. Ví dụ, auto fire.
* **bool GetKeyDown(int keyCode)**: Trả về true khi một phím được nhấn.
* **bool GetKeyUp(int keyCode)**: Trả về true khi một phím được nhả ra sau khi nhấn.

### Class Sound

Trong DirectX có một component DirectSound chuyên xử lý việc hỗ trợ phát âm thanh trong game. Class này đơn giản chỉ cần các tính năng khởi tạo, load, phát âm thanh wave (có phần mở rộng .wav).

Trong DirectX SDK có một thư viện DXUTsound. Dĩ nhiên thư viện này cũng hỗ trợ phát âm thanh. Các lớp và phương thức có liên quan đến việc phát âm thanh (như CSoundManager, CSound, CWaveFile) có một số vấn đề trong các phiên bản khác nhau của DirectX, chủ yếu ở việc tổ chức lại các lớp này. Cho nên để tránh rườm rà, Jonathan S.Harbour (tác giả của quyển sách Beginning Game Programming) đã tự viết riêng một cặp file: DirectSound.h, DirectSound.cpp. Trong phạm vi đồ án xin phép sử dụng lại thư viện của tác giả.

Trong thư viện có 3 class:

* **CSoundManager:** thiết bị DirectSound chính
* **CSound:** Được sử dụng để tạo các DirectSound đệm (buffers)
* **CWaveFile:** Được dùng để load các file âm thanh .wav lên buffer của CSound.

Đó là các class phục vụ cho mọi việc bên dưới hệ thống, tác giả đã cung cấp cho ta. Còn việc gọi các hàm thao tác với file âm thanh (có thể nghĩ ngay đến hàm Play, Loop, Stop, …) thì ta viết thành một class riêng. Ở đây đặt là Sound.h và Sound.cpp

Đầu tiên, để sử dụng được DirectSound, chúng ta cần khởi tạo bằng hàm:

* **bool DirectSound\_Init(HWND hwnd):** Hàm này tạo ra một object thuộc class CSoundManager, gọi hàm Initialize, đặt format cho buffer chính.
* **void DirectSound\_Shutdown():** Ngược với hàm Init, hàm này xóa đi con trỏ đã khởi tạo ở hàm Init.
* **void LoadSound(string filename):** Hàm này được gọi ở các hàm Create của các object “có” âm thanh. Trong này sẽ có một buffer thứ hai \*wave.
* **void Play(CSound \*sound):** Phát âm thanh.
* **void Loop(CSound \*sound):** Phát âm thanh liên tục.
* **void Stop (CSound \*sound):** Dừng phát âm thanh.

3 hàm trên gọi trực tiếp từ thư viện DirectSound.

### 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/>

#### 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.

#### 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.

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).

#### 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

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.

#### 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.

## **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.

# **Game Metroid:**

### Intro Scene:

Intro Scene là một scene xuất hiện khi vừa vào game. Phần này chủ yếu giới thiệu về thông tin game như tên game, bối cảnh, năm sản xuất,… Đặc điểm của Intro Scene là nó phát đi phát lại hình ảnh kèm âm thanh cho đến khi nhận được phím Enter.

Với mô tả như trên, ta có thể xem Intro Scene là một tập hợp các Sprite và phát kèm âm thanh nền. Ở đây, nhóm đã dùng một chút kỹ thuật để làm Intro Scene. Đó là sử dụng một ảnh gif đã có sẵn trên trang web [www.pinterest.com](http://www.pinterest.com) để lấy một ảnh gif có nội dung là Intro Scene. Sau đó sử dụng trang [www.piskel.com](http://www.piskel.com) để tách ảnh gif ra thành một spritesheet. Kết quả cho ra 169 sprite gồm 13 dòng 13 cột!!! Dĩ nhiên không thể nhập vào từng dòng đến hết 169 sprite vào file .xml để mô tả hết sprite này. Việc này để cho một chương trình nhập xuất cơ bản có sử dụng vòng lặp thực hiện, việc còn lại là copy kết quả vào file .xml đặc tả Intro Scene.

Sau khi chèn thêm âm thanh nền, ta đã có một Intro Scnene.

### Menu Scene:

Menu Scene là scene tiếp theo của Intro Scene. Mô tả: Sau khi bắt được phím Enter, Intro Scene sẽ đổi sang 2 Sprite của Menu Scene. Khi bấm nút mũi tên lên (DIK\_UP) và xuống (DIK\_DOWN) sẽ chuyển sang các Sprite tương ứng. Các Sprite này cũng được mô tả trong một file .xml

### PlayScene:

#### Map:

Với sự hỗ trợ của Tiled Map Editor và lớp TMXMap có trong framework, việc render map trở nên khá thuận tiện. Đầu tiên, chúng ta load map cần render bằng class MapLoader:

//load map bằng maploader

mapLoader.AddMap("map1", "Resources/map3.tmx", 1);

//trả về con trỏ đến map mong muốn

map = mapLoader.GetMap("map1");

Và tại hàm Update của PlayScene, để thực hiện render map, ta chỉ cần:

//render map

map->Render(batch);

#### Player:

Đối với mỗi đối tượng có khả năng va chạm và di chuyển như Player, ngoài việc chứa một sprite (hoặc kế thừa từ Sprite hoặc một số lớp khác như Texture, TextureRegion), player cũng cần thiết chứa một Body để việc biểu diễn va chạm trở nên dễ dàng.

//setup body

BodyDef bodyDef;

bodyDef.bodyType = Body::BodyType::Dynamic;

bodyDef.size.Set(30, 60);

body = world->CreateBody(bodyDef);

body->categoryBits = PLAYER\_BIT;

body->maskBits = SKREE\_BIT | ZOOMER\_BIT |PLATFORM\_BIT;

Trên đây chỉ là một ví dụ về việc setup body cho player. Đương nhiên ở trong game, player có thể va chạm với rất nhiều đối tượng khác trong World, điều này được thực hiện bằng việc bổ sung các bit cho body của player.

Ngoài ra khi ở trong game, player không thể thực hiện hành động nhảy một cách vô tận khi nhận input mà chỉ có thể thực hiện hành động này của mình khi đã tiếp đất. Để xử lý vấn đề này, ta có thể tạo thêm một Body cho lớp Player:

//create foot

BodyDef footDef;

footDef.bodyType = Body::BodyType::Kinematic;

footDef.size.Set(30, 15);

footDef.isSensor= true;

foot = world->CreateBody(footDef);

foot->categoryBits = FOOT\_BIT;

foot->maskBits = PLATFORM\_BIT;

foot->SetPosition(body->GetPosition().x, body->GetPosition().y - 30);

Mục đích của body này là xác định việc player đã chạm đất hay chưa. Nếu đã chạm đất, ta cho phép player nhận input và thực hiện hành động nhảy.

#### Zoomer:

Zoomer có cách di chuyển khá là đặc biệt. Nó di chuyển bằng cách bám vào 1 mặt phẳng và di chuyển dọc theo mặt phẳng đó.

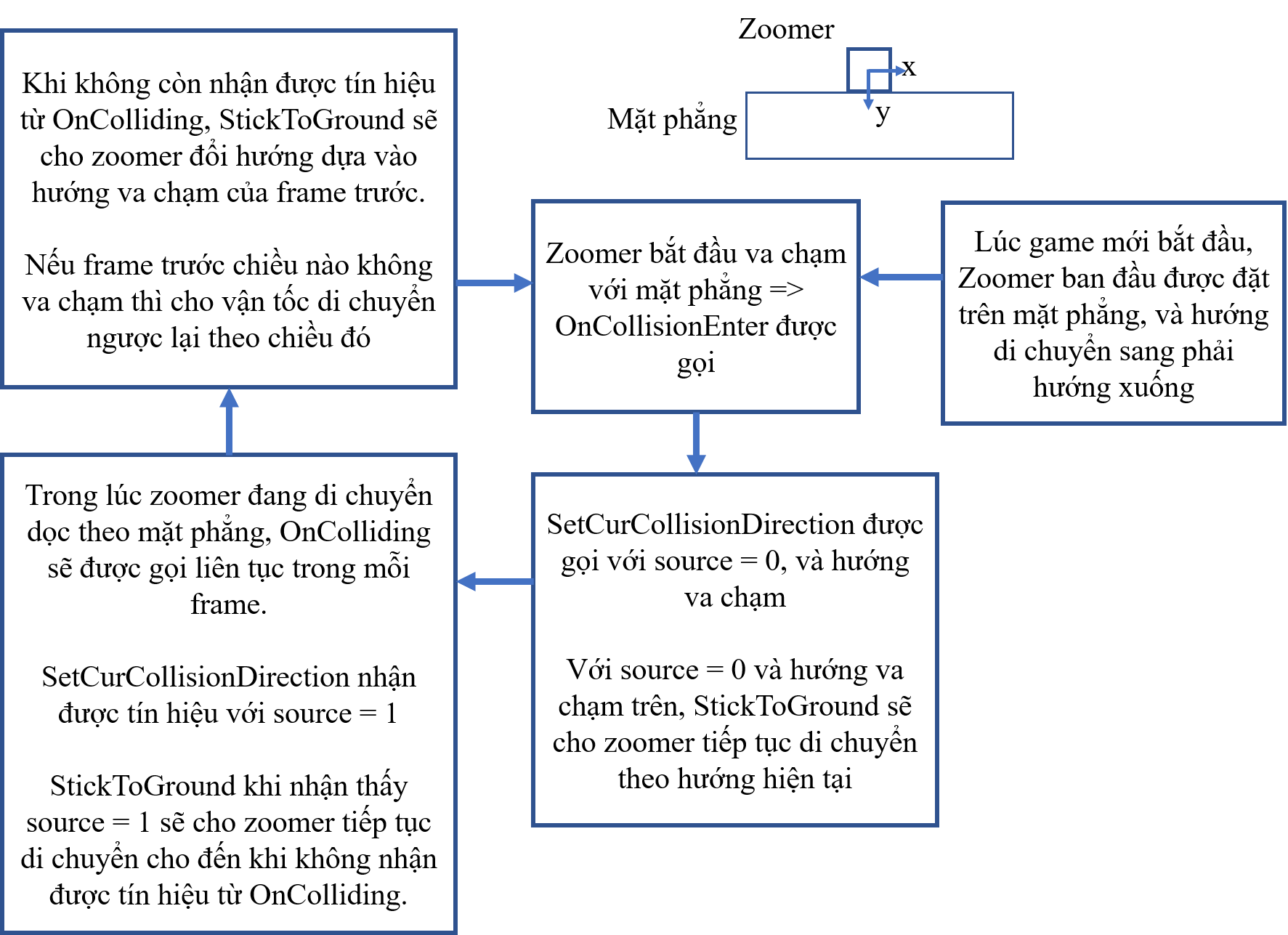
Trong lớp Zoomer, sẽ có 2 phương thức chính sẽ được dung để khiến cho zoomer di chuyển bám vào mặt phẳng:

* **StickToGround()**: kiểm tra và quyết định hướng đi tiếp theo của zoomer trong từng frame.
* **SetCurCollisionDirection(Vector2 collisionDirection, int source)**: lấy hướng va chạm giữa zoomer với 1 vật thể khác, và nguồn đã lấy hướng va chạm.

Phương thức **SetCurCollisionDirection** có thể được gọi từ 1 trong 2 nguồn sau:

* **Nguồn 0 (source = 0)**: được gọi từ hàm OnCollisionEnter trong WorldListener. Có nghĩa là zoomer bắt đầu va chạm với 1 vật thể nào đó
* **Nguồn 1 (source = 1)**: được gọi từ hàm OnColliding trong WorldListener. Có nghĩa là zoomer đang va chạm với 1 vật thể nào đó

Sơ đồ trình tự thực hiện zoomer:



#### Ripper:

Ripper là một Enemy khá dễ trong số các Enemy của Brinstar. Mô tả: Ripper chỉ di chuyển theo phương ngang Ox với một vận tốc đặt trước, khi Ripper va chạm với Platform, Ripper sẽ quay đầu, và đảo chiều vận tốc. Đặc trưng của Ripper là không bị ảnh hưởng bởi các loại đạn thường.

Class Ripper bên cạnh các hàm Create, Render, Update thì còn có các hàm đặc trưng như sau:

* **void OnHitGround ():** Xử lý khi phát hiện va chạm giữa Ripper và Platform. Ở đây cụ thể của hàm này ta tiến hành gọi hàm ChangeDirection (để quay đầu lại) và báo hiệu cho hàm Update biết rằng Ripper đã quay đầu, đề nghị, đổi lại chiều chuyển động cho nó.
* **void OnHitBullet ():** Vì Ripper không ảnh hưởng bởi đạn thưởng của player, nên ở phạm vi này, ta không cần xử lý nhiều khi có va chạm giữa Ripper và đạn của player. Khi 2 object chạm nhau, chỉ việc phát ra âm thanh hiệu ứng là xong.

#### Door

Trong Metroid có các Door (cũng có thể gọi là Hatch), có chức năng ngăn các không gian với nhau. Chỉ khi player đi qua khỏi Door thì mới có thể nhìn thấy không gian tiếp theo. Mô tả: Door gồm 3 phần, 2 phần màu xanh trái và phải được đặt là Left và Right, phần giữa được đặt là Mid. Left hoặc Right chỉ mở ra khi có va chạm với ít nhất một viên đạn của player. Nếu Player trong khoảng thời gian cửa left hoặc right mở mà chạm vào Mid, thì sẽ có hiệu ứng chuyển đến không gian kế tiếp. Sau đó left/right door lập tức đóng lại khi player đã đi qua. Nếu player không đi qua, thì sau một thời gian đóng lại. Các door chỉ cho phép player đi qua, không cho phép bất cứ thứ gì khác từ các Enemy, Boss đến đạn Bullet. Các door chuyển đến không gian có chứa boss thì đặc biệt hơn, chỉ khi đã giết được boss, người chơi mới có thể mở và trở về.

Như đã mô tả, một door được set vị trí trên tile map là vị trí của Mid, hai vị trí Left Right chỉ cần thêm bớt một hằng số so với tọa độ của Mid.

Trong class Door có các hàm đặc trưng sau:

* **void LOnhitBullet() / void ROnhitBullet()**: Xử lý khi phát hiện cửa trái/phải va chạm với đạn Bullet.
* **void SetLeftOpen (bool state) / void SetRightOpen(bool state):** Điều khiển cửa trái phải đóng hoặc mở
* **bool IsLOpen()/bool IsROpen():** kiểm tra xem cửa trái/phải đã đóng hay mở.
* **void SetCanPassLeft(bool flag)/void SetCanPassRight(bool flag):** cho phép player được qua trái/phải của Door.
* **bool GetCanPassLeft()/bool GetCanPassRight():** kiểm tra xem player được phép qua cửa chưa, nếu được tiến hành mở hiệu ứng qua cửa.

Giả sử Player đi từ trái sang phải, chương trình chạy tuần tự như sau:

Sau khi khởi tạo, Player bị giữ lại ở cửa trái. Khi bắn, bullet va chạm với door, lúc này đi vào hàm LOnhitBullet. Trong hàm này ra lệnh bật cờ hiệu isLOpen = true để ra hiệu cho Update ngưng update cửa, và chèn âm thanh vào. Lúc này cửa trái sẽ “mở” (biến mất và maskbit của Door không còn PLAYER\_BIT nữa). Lúc này một biến đo thời gian pauseTimeL sẽ hoạt động, đến một giá trị nào đó, cửa trái sẽ tự động đóng lại, tức là hạ cờ isLOpen = false. Nếu trước khi hạ cờ, player chạm được phần Mid của cửa, sẽ đi vào hàm SetCanPassRight(true), tức là chuyển đến hiệu ứng “qua cửa”. Hiệu ứng qua cửa gồm thứ tự như sau:

-Đóng cửa trái lập tức (bit cửa trái trở về giá trị ban đầu)

-Mở cửa phải.

-Mở cửa Mid (không nhận va chạm với Player nữa)

-Set vận tốc cho camera trong class PlayScene.

-Dừng Update Player, không nhận phím điều khiển di chuyển từ bàn phím.

-Set vận tốc cho Player

-Sau khi Player qua cửa (dùng một biến passTime để đo thời gian), đóng cửa phải, SetCanPassRight(false), biến PassTime trả về 0.

Lưu ý ở mỗi lần đóng và mở cửa đều có âm thanh hiệu ứng.

### GameOverScene:

GameOverScene là một scene sau khi phát hiện player “chết” (health <=0). Khi player chết, PlayScene sẽ dừng một khoảng thời gian, sau đó sẽ chuyển đến Sprite GameOver đã được mô tả trong file .xml.

# **Kết luận:**

## **Kết quả đạt được:**

* **Ôn lại và củng cố kiến thức từ các môn lập trình căn bản học kì trước.**

Các kiến thức nền tảng từ Lập trình hướng đối tượng như kế thừa, đa hình,…

Sử dụng các kĩ thuật lập trình cơ bản từ Nhập môn lập trình như

các câu điều kiện,…

* **Kỹ năng phân tích giải quyết vấn đề**Phân tích, tìm hiểu “gốc rễ” của một vấn đề từ đó đưa ra những phương án giải quyết.

Tính toán, đúc kết được phương án tối ưu.

* **Kỹ năng làm việc nhóm**Phân chia công việc phù hợp với từng thành viên.

Rèn luyện và xây dựng khả năng giao tiếp với mọi người.

Truyền tải được thông điệp ý tưởng, những đề xuất khi làm việc nhóm.

Hình dung được quá trình làm việc của một nhóm.

Biết thông cảm, giúp đỡ, hỗ trợ lẫn nhau khi một thành viên trong nhóm gặp khó khăn.

Học được cách viết code cho người khác hiểu và nắm bắt được.

* **Kỹ năng quản lí mã nguồn**

Tiếp cận được cách quản lí mã nguồn code.

Biết được cách tổ chức code, qui định về style code, comment code để không làm khó các thành viên còn lại.

## **Hướng cải tiến và phát triển:**

* Hoàn thành toàn bộ game
* Tối ưu hoá việc sử dụng tài nguyên máy

# **Tài liệu tham khảo:**

<https://libgdx.badlogicgames.com/>

<http://box2d.org/>

# **Bảng phân công công việc**

|  |  |
| --- | --- |
| Trương Ngọc Sơn | SpriteBatch, PlayScene, Camera, Game, Scene, TMXMap |
| Nguyễn Đình Thiện | Collision, Zoomer, World, Body, boss Kraid |
| Trần Minh Trí | CollisionQuadTree, Rio, Collision, World, boss Mother Brain |
| Trần Phú Vinh | Sound, Font, Label, Sprite, Texture, TextureRegion, Door, Ripper |