Đề tài: Lập trình game Run!!

**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 Run!! được dựa trên ý tưởng của game Geometry Dash. Là một tựa game rhythm-based platformer, Game cho phép người chơi vào nhân vật hình vuông, sử dụng one-touch gameplay, người chơi điều khiển nhân vật qua các chướng ngoại vật đến khi về đích

**(Ảnh game)**

Ngoài ra Game Run!! còn hỗ trợ multiplayer với tối đa 4 người cùng chơi.

**(Ảnh Multiplayer trong game)**

2. Công nghệ sử dụng:

Android Studio: Là một môi trường phát triển tích hợp (IDE) cho phép người dùng phát triển các ứng dụng cho điện thoại android một cách nhanh chóng, dễ dàng.

LibGDX: Là một framework phát triển game cả 2D và 3D, được viết bằng ngôn ngữ lập trình Java. LibGDX cho phép phát triển game trên nhiều nền tảng khác nhau mà chỉ cần 1 lần viết code.

Tiled Map Editor: Là một công cụ giúp người dùng vẽ các map trong game (Đặc biệt là tile map) dễ dàng hơn.

Firebase: Là một dịch vụ cơ sở dữ liệu thời gian thực hoạt động trên nền tảng đám mây được cung cấp bởi Google nhằm giúp các lập trình viên phát triển nhanh các phần back-end của ứng dụng như Database, Hosting, Authentication, …

Node.js: Là một nền tảng mã nguồn mở cho phép xây dựng các ứng dụng liên quan đến network và quản lý server bằng ngôn ngữ JavaScript.

Socket.io: Là một thư viện chạy trên Node.js cho phép xây dựng dễ dàng hơn ứng dụng có sự giao tiếp 2 chiều real-time giữa các client.

3. Mục tiêu:

Ngoài mục đích học thêm về Android Studio và ngôn ngữ Java, việc xây dựng ứng dụng còn giúp nhóm học thêm về những khái niệm căn bản có trong game, cách game hoạt động với sự hỗ trợ của LibGDX framework. Việc xây dựng phần chơi multiplayer của game còn giúp nhóm có một khái niệm căn bản về việc xây dựng server-client.

II. Thiết kế:

1. Kiến trúc chương trình:

Game sẽ được chia các màn hình game được gọi là Screen và được quản lý bởi một lớp Game Manager duy nhất.

Các Screen cơ bản:

* Menu Screen
* Map Selection Screen
* Play Screen

Ngoài ra còn một số Screen khác tuy nhiên ở đây để phục vụ cho mục đích dễ dàng hiểu rõ hơn kiến trúc chương trình, chúng ta sẽ xét hoạt động của 3 Screen trên.

Cách thức game hoạt động và các Screen cơ bản được mô tả như hình bên dưới:

Các lệnh được gọi

Người dùng kết thúc game

Game được kết thúc

Bắt đầu game

Game Manager

Menu Screen

Map Selection Screen

Play Screen

Các Screen có thể chuyển đổi trực tiếp

Lớp Game quản lý việc thực hiện việc giữ cho game hoạt động và chuyển đổi giữa các Screen với nhau. Ví dụ, khi bắt đầu game, người dùng ở Menu Screen. Khi người dùng đang ở Menu Screen và muốn chuyển sang Map Selection Screen, người dùng bấm nút SinglePlayer. Lập tức, Menu Screen gửi một lệnh chuyển đổi Screen cho Game Manager và Game Manager thực hiện chuyển đổi sang Map Selection Screen.

2. Phân tích chương trình:

Trước khi đi vào phân tích cụ thể game Run, 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. LibGDX đã cung cấp cho chúng ta đủ mọi thứ để có thể làm một game hoàn chỉnh.

2.1. Các đối tượng chính cần biết của LibGDX:

2.1.1. Class Game:

Lớp Game cung cấp bởi LibGDX là một lớp ảo và được implement từ ApplicationListener.

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

* **create():** Hàm này được gọi 1 lần duy nhất ngay khi game vừa bắt đầu.
* **render():** Được gọi 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 render.
* **dispose():** Giải phóng các tài nguyên cần thiết để tránh memory leak như texture, sound, … Việc gọi hàm này được LibGDX tự động gọi khi người dùng tắt game ứng dụng.

Ngoài ra LibGDX còn cung cấp cho lớp Game này một số hàm hữu dụng khác như:

* **pause():** Được gọi khi game bị dừng. Thường là khi game không được kích hoạt hoặc là không được nhìn thấy trên màn hình.
* **resume():** Được gọi khi game được tiếp tục từ trạng thái đang pause() ở trên.
* **resize(int width, int height):** Được gọi khi game bị thay đổi kích thước màn hình.
* **setScreen(Screen screen):** Chỉ cho lớp Game biết Screen nào nên được render. Chi tiết hơn về hàm này sẽ được nói thêm ở các mục bên dưới.

Hàm **setScreen(Screen screen)** là một hàm rất quan trọng trong việc quản lý các Screen vì vậy chúng ta cần phải lưu ý tận dụng hàm này.

2.1.2. Interface Screen:

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. Vì vậy, LibGDX cung cấp sẵn interface Screen để việc quản lý các cảnh game dễ dàng hơn.

Các hàm căn bản nhất của Interface Screen là:

* **render(float delta):** Tương tự như lớp Game, hàm này được gọi mỗi frame và mọi hoạt động của game được diễn ra tại đây.
* **dispose():** Giải phóng các tài nguyên đã cấp.

Các hàm căn bản của interface Screen 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 GameManager thực hiện việc render cho lớp MenuScreen ở hàm render(). Cách lớp Game thực hiện render cho Screen sẽ được nói rõ hơn khi ta đi vào phân tích Game Run!! cụ thể.

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

2.1.3. 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(FileHandle file):** constructor để khởi tạo texture với một file hình ảnh đã có.
* **dispose():** Giải phóng texture này.

Đây là 2 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 và chiều rộng của bức ảnh thì LibGDX cũng có cung cấp một vài hàm khác như getWidth(), getHeight(), …



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

Texture sẽ không được tự động giải phóng mà chúng ta phải quyết định lúc nào nên giải phóng texture này. Lý do tương tự như lớp Screen là 1 texture có thể được sử dụng lại nhiều lần để vẽ.

2.1.4. 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 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.



Hình 2 SpriteSheet và tọa độ các nhân vật trong SpriteSheet



Hình 3 Hình ảnh khi vẽ lên màn hình

Các hàm căn bản của TextureRegion là:

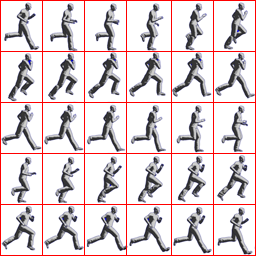
* **TextureRegion(Texture texture):** Constructor với một texture.
* [**TextureRegion**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/TextureRegion.html#TextureRegion-com.badlogic.gdx.graphics.Texture-float-float-float-float-)**(**[**Texture**](https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/Texture.html)**texture, int x, int y, int width, int height):** 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.
* **setRegion(int x, int y, int width, int height):** hàm set 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:

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.



Hình 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:

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:

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:

2.1.9: Class Viewport:

2.1.10. Class Body:

2.1.9: Class World:

2.1.10: Class ContactListener

2.1.11: Class Stage:

2.1.12: Class TextButton:

2.1.13: Input:

2.2. Các đối tượng chính trong game Run:

2.2.1. Lớp GameManager:

**Extends class Game**

Được kế thừa từ lớp **Game** từ LibGDX framework, như đã nói ở trên, **GameManager** thực hiện việc quản lý các Screen và các Resource cần thiết.

3 hàm căn bản của lớp **GameManager** kế thừa từ lớp Game là:

* **create():** Hàm này được gọi 1 lần duy nhất ngay khi game vừa bắt đầu.
* **render():** Được gọi 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 render.
* **dispose():** Giải phóng các tài nguyên cần thiết để tránh memory leak như texture, sound, …

Khi bắt đầu ứng dụng, lớp GameManager thực hiện việc tạo ra 1 **SpriteBatch** và các **Screen** cần thiết.

Phải nói thêm, việc tạo ra 1 SpriteBatch ở mỗi lớp Screen để thực hiện vẽ các đối tượng là hoàn toàn có thể. Tuy nhiên, việc này gây ra việc tiêu tốn tài nguyên vì thực sự chúng ta chỉ cần dùng 1 SpriteBatch cho toàn bộ game. Vì vậy, tạo ra 1 SpriteBatch ở GameManager là hợp lý.

Tạo ra 1 SpriteBatch rất đơn giản, chúng ta chỉ cần gọi:

batch = new SpriteBatch();

Lệnh này nên được thực hiện ở hàm create() của lớp GameManager.

Các Screen căn bản cũng được tạo tại hàm create() của lớp này:

menuScreen = new MenuScreen(this);  
mapSelectionScreen = new MapSelectionScreen(this);

Các lớp screen này sẽ được giải thích rõ hơn ở các phần sau.

Bởi vì game sẽ bắt đầu với MenuScreen, sau khi khởi tạo các lớp và tài nguyên cần thiết thì tại hàm **create()**, ta có thể thực hiện:

setScreen(menuScreen);

**setScreen(Screen screen)** là một hàm được cung cấp sẵn bởi lớp Game cho phép GameManager biết được nên render Screen nào.

Ngoài ra, việc quản lý một lúc nhiều hình ảnh và âm thanh khá là rắc rối và hơn nữa, có một vài tài nguyên chúng ta có thể sử dụng lại cho nhiều Screen khác nhau, LibGDX cung cấp lớp đối tượng **AssetManager** cho phép ta quản lý các tài nguyên này một cách dễ dàng hơn.

Trong hàm **create()**:

assetManager = new AssetManager();

Và việc load các tài nguyên được thực hiện như sau:

assetManager.load(String fileName, Class<T> type);

fileName là đường dẫn đến tài nguyên và type là loại tài nguyên (Có thể là Texture.Class, Sound.Class, …)

2.2.2. Lớp MenuScreen

**Implements interface Screen**

Các hàm căn bản của lớp **MenuScreen** nói riêng và các lớp implement từ **Screen** nói chung là:

* constructor: Khởi tạo các lớp, các tài nguyên cần thiết.
* render(float delta): Tương tự như lớp GameManager, hàm này được gọi mỗi frame và mọi hoạt động của game được diễn ra tại đây.
* dispose(): Giải phóng các tài nguyên đã cấp.

Các hàm căn bản của lớp MenuScreen gần như không có gì quá khác biệt với các hàm căn bản của lớp GameManager. Điều này là dễ hiểu vì thực chất, lớp GameManager thực hiện việc render cho lớp MenuScreen ở hàm render().

@Override  
public void render () {  
 super.render();  
 ...

Một lỗi thường thấy ở việc viết hàm render() cho lớp GameManager là việc bỏ qua super.render(). Thực chất, hàm này thực hiện việc gọi hàm render(float delta) của lớp Screen hiện tại.

Interface Screen không cung cấp phương thức create() như là lớp Game, vì vậy chúng ta tạo ra constructor để khởi tạo các tài nguyên mà chúng ta mong muốn.

public MenuScreen(GameManager gameManager)

constructor của lớp MenuScreen nhận vào một đối số là **GameManager**. Như đã phân tích ở lớp Game, việc sử dụng lại **SpriteBatch** là dư thừa và chúng ta cũng cần phải lấy các tài nguyên cần thiết từ **AssetManager.** Vì vậy, chúng ta truyền vào đối số GameManager để giúp chúng ta lấy ra được những đối tượng này.

Lớp MenuScreen thực hiện việc render các button như SinglePlayer button, MultiPlayer button ChangeCharacter button, … Các button này không có gì quá đặc biệt. Nó chỉ thực hiện việc thông báo cho GameManager biết để chuyển sang Screen tương ứng.

Ví dụ: SinglePlayer button

Đầu tiên để button này có thể nghe được sự kiện người dùng nhấn vào button, ta cần thêm một InputListener:

singlePlayerButton.addListener(new InputListener() {  
 @Override  
 public boolean touchDown(InputEvent event, float x, float y, int pointer, int button) {

...  
 return true;  
 }  
});

Và tại hàm touchDown, để chuyển sang một Screen khác, ta làm như sau:

gameManager.setScreen(gameManager.getMapSelectionScreen());

Để vẽ button này lên màn hình game, ta cần một **Stage**.

menuViewPort = new StretchViewport(*WORLDWIDTH*, *WORLDHEIGHT*);  
stage = new Stage(menuViewPort, gameManager.batch);

WorldWidth và WorldHeight là chiều dài và chiều rộng của màn hình game (Nói đúng hơn là của camera)

Ngoài ra, LibGDX cũng cung cấp lớp đối tương **Table** giúp cho việc đặt các button trở nên dễ dàng hơn mà không cần phải tính toán quá nhiều về vị trí để đặt button cho thích hợp.

table = new Table();  
table.center();  
table.setFillParent(true);

//Thêm button vào table

table.add(singlePlayerButton);

//Thêm table vào stage

stage.addActor(table);

Để vẽ stage, ta chỉ cần gọi:

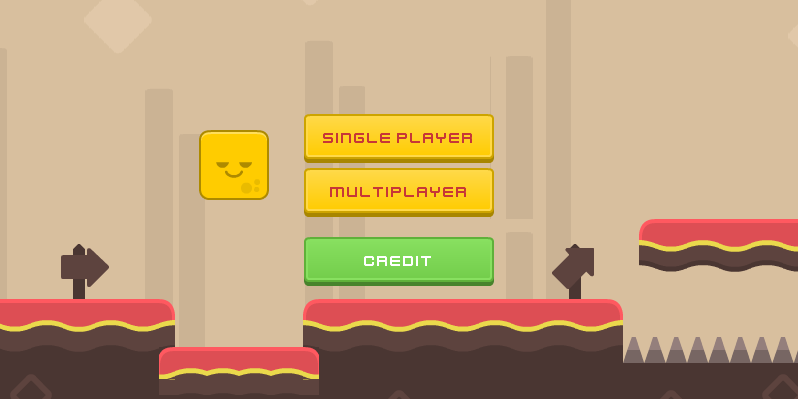
stage.draw();

Hàm này được thực hiện tại hàm **render(float dt)** của MenuScreen.

Một điều mà chúng ta thường hay quên nữa là việc giải phóng tài nguyên cho stage. Tại hàm **dispose()** của lớp MenuScreen:

stage.dispose();

Dưới đây là hình ảnh Screen này sau khi hoàn thành:

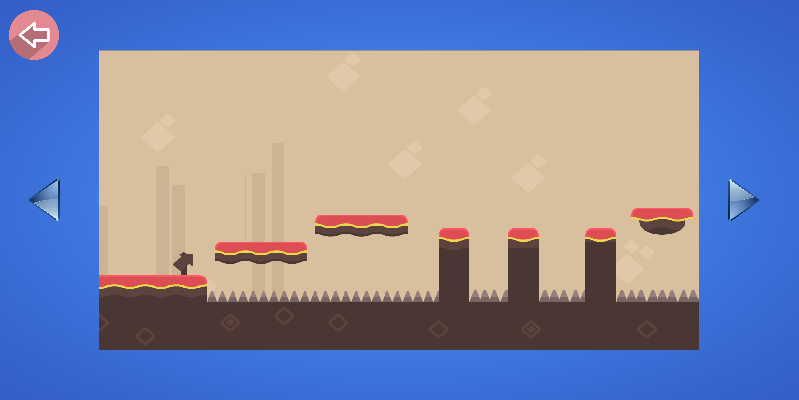


Hình 5 MenuScreen

2.2.3. Lớp MapSelectionScreen

**Implements class Screen**

Có lẽ không có gì quá nhiều để nói về lớp này bởi vì mục đích của nó hoàn toàn giống với MenuScreen là hiển bị một vài button cần thiết.



Hình 6 MapSelectionScreen

2.2.4. Lớp PlayScreen

**Implements class Screen**

Đây là lớp Screen quan trọng nhất của game bởi vì sẽ chả ai chơi game nếu không có thứ gì để chơi!!!



Hình 7 Một màn chơi trong game

Lớp PlayScreen là lớp khá phức tạp và chứa rất nhiều đối tượng. Vì vậy, chúng ta sẽ đi vào phân tích một vài đối tượng chính cụ thể trước khi đi vào phân tích lớp PlayScreen này.

2.2.4.1. Camera và Viewport:

Camera trong game đóng vai trò là một cửa sổ cho phép chúng ta nhìn vào một phần nhỏ của một thế giới game rộng lớn. Hơn nữa, việc thay đổi vị trí Camera giúp cho ta có thể nhìn vào các phần khác nhau của thế giới game.

Lưu ý: Phần bài viết dưới đây nói về màn hình và màn hình game. Hai khái niệm này khá là dễ nhầm lẫn. Khi nói đến màn hình thì chúng ta ý nói đến màn hình của điện thoại hoặc màn hình máy tính. Ví dụ, màn hình điện thoại có nhiều kích cỡ. Còn khi nói đến màn hình game, chúng ta ám chỉ đến kích thước viewport.

Bước khởi tạo Camera khá đơn giản:

mainCamera = new OrthographicCamera();

Bởi vì game Run là một game 2D vì vậy chúng ta sẽ sử dụng OrthographicCamera. LibGDX còn cung cấp PerspectiveCamera cho việc render game 3D tuy nhiên chúng ta sẽ không thảo luận Camera này ở đây.

Viewport hiểu đơn giản là kích cỡ của camera. Hơn nữa, khi kích thước màn hình thay đổi, viewport giúp ta luôn giữ một tỉ lệ màn hình game tương đối.

gameViewPort = new StretchViewport(worldWidth, worldHeight, mainCamera);

Một việc mà chúng ta thường quên đó là tại hàm **resize(int width, int height)** của lớp PlayScreen:

gameViewPort.update(width,height);

Điều này đảm bảo cho việc khi kích thước màn hình thay đổi thì hàm **resize** sẽ được gọi và viewport cũng sẽ thay đổi phù hợp theo phù hợp với tỉ lệ màn hình.

2.2.4.2. World và Body:

World chứa các Body và thực hiện biểu diễn va chạm giữa các Body.

world = new World(new Vector2(0f, -9.8f), true);

Để việc biểu diễn va chạm được diễn ra, ta chỉ cần gọi:

world.step(1 / 60f , 6, 2);

2.2.4.3. Class Player:

**Extends class Sprite**

Để tạo một hình ảnh cho Player, ta có thể sử dụng hàm đã được cung cấp sẵn như sau:

setTexture("Player.png");

Vì player có thể va chạm với các vật thể khác trong thế giới game, vì vậy lớp này cần chứa một body. Việc khởi tạo body đã được giới thiệu rất kĩ trong các tài liệu của LibGDX, tuy nhiên chúng ta cũng nên nhắc lại nó một chút:

//Khởi tạo BodyDef là lớp định nghĩa cho một Body

BodyDef bDef = new BodyDef();  
bDef.position.set(this.getX()+this.getWidth()/2,this.getY()+this.getHeight()/2);  
bDef.type = BodyDef.BodyType.*DynamicBody*;  
body = world.createBody(bDef);  
  
//Tạo hình dáng của Body với lớp FixtureDef   
FixtureDef fDef = new FixtureDef();  
PolygonShape bodyShape = new PolygonShape();  
bodyShape.setAsBox(this.getWidth()/2,this.getHeight()/2);  
fDef.shape = bodyShape;

body.createFixture(fDef).setUserData(this);

Việc **setUserData(this)** (**this** ở đây chính là lớp Player) là rất quan trọng. Còn tại sao nó lại quan trọng thì chúng ta sẽ nói rõ hơn ở các mục sau.

Để phục vụ cho việc cập nhật vị trí của player đúng với vị trí của Body sau mỗi frame, ta tạo ra hàm **update(float dt)** và hàm này được gọi tại hàm **render(float dt)** của lớp PlayScreen

public void update(float dt) {  
 //update texture position  
 setPosition(body.getPosition().x - getWidth() / 2,

body.getPosition().y - getHeight() / 2);  
  
}

Đối với tựa game Run này, ta sẽ cho player tự động chạy về phía bên phải và khi người dùng chạm màn hình ta sẽ cho nhân vật nhảy lên khỏi mặt đất.

//Luôn đi sang phải với vận tốc lớn nhất là 2f  
if(body.getLinearVelocity().x < 2f) {  
 body.applyLinearImpulse(0.1f, 0f, body.getPosition().x, body.getPosition().y, true);  
}  
//Nhảy lên khi người chơi chạm vào màn hình  
if (Gdx.*input*.justTouched()) {  
 body.setLinearVelocity(body.getLinearVelocity().x, 4f);  
}

2.2.4.4. Ground:

2.2.4.5. WorldListener:

2.2.4.6. MapCreator:

3. Các tính năng bổ sung:

3.1. Đăng nhập với Firebase:

Trong một tựa game hỗ trợ multiplayer, việc cho phép người dùng đăng nhập tài khoản để ghi nhớ thứ hạng trong game, quá trính chơi game, các điểm số, … là rất cần thiết. Tuy nhiên, việc tạo ra hệ thống đăng nhập này là không hề đơn giản. Hiểu được điều này, Firebase cung cấp các công cụ cần thiết để giúp các nhà phát triển dễ dàng trong việc xây dựng hệ thống này.

Việc tích hợp Firebase cho ứng dụng sử dụng Android Studio đã được hướng dẫn rất chi tiết tại trang chủ của Firebase.

**Setup*:*** [*https://firebase.google.com/docs/android/setup*](https://firebase.google.com/docs/android/setup)

**Google Sign-in:** [*https://firebase.google.com/docs/auth/android/google-signin*](https://firebase.google.com/docs/auth/android/google-signin)

**Facebook Login:** [*https://firebase.google.com/docs/auth/android/facebook-login*](https://firebase.google.com/docs/auth/android/facebook-login)

3.2. Server:

Node.js và Socket.io là các công cụ cần thiết để tạo lập server nhằm cung cấp môi trường và tạo lập kết nối giữa các người chơi.

Bởi vì tài liệu về tích hợp Node.js và socket.io cho Android Studio là khá ít và rời rạc, nội dung dưới đây sẽ giới thiệu sơ qua về cách thực hiện việc này.

Về các khái niệm và cách sử dụng Node.js và socket.io:

**Node.js:** <https://nodejs.org/en/docs/>

**Socket.io:** <https://socket.io/get-started/chat/>

Để tích hợp socket.io cho Android Studio, tại file build.gradle ở phần dependencies của project(“:core”):

compile "io.socket:socket.io-client:1.0.0"

Khởi tạo một file index.js và để bắt đầu khởi tạo server:

//khởi tạo application instance

var app = require('express')();  
var server = require('http').Server(app);

//Được gọi mỗi khi có một request đến server   
app.get('/', (req, res) => res.send('Welcome to Run Game Hosting!!!'));  
//Set server port 5000  
app.set('port',5000);

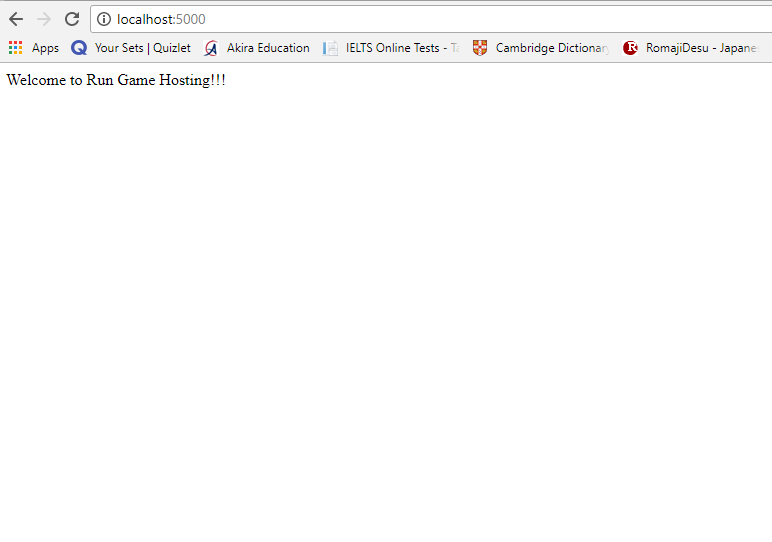
//Nghe các request tại port 5000)  
server.listen(app.get('port'), function(){

//server sẵn sàng để nghe các request   
 console.log("Server is now running...");  
});

Để chạy file index.js, ta sử dụng node command:

node index.js

Và đây là kết quả:



Để sử dụng socket.io, tại index.js:

var io = require('socket.io')(server);

io.on('connection',function(socket){

console.log("Player Connected!");

}

Giải thích một chút về dòng lệnh **io.on(..):** thực chất dòng lệnh này hoạt động như một event listener. Khi người dùng vừa kết nối vào server, server nhận biết điều này và phát ra một event ‘connection’ và Event này được socket.io lắng nghe.

Và tại Android Studio, để thực hiện công việc tương tự:

//khởi tạo socket io và kết nối đến server  
socket = IO.*socket*("http://localhost:5000");  
socket.connect();

//Lắng nghe sự kiên connection  
socket.on(Socket.*EVENT\_CONNECT*, new Emitter.Listener() {  
 @Override  
 public void call(Object... args) {  
 Gdx.*app*.log("SocketIO", "Connected");  
 }  
});

Việc gửi và nhận tín hiệu giữa server và client được thực hiện thông qua các event. Để dễ hiểu, ta lấy ví dụ về việc gửi vị trí player của một client đến các client khác:

Tại Android Studio, khi player thay đổi vị trí:

//Khởi tạo object để chứa các data cần thiết

JSONObject data = new JSONObject();  
try {  
 data.put("x", player.getX());  
 data.put("y", player.getY());  
 socket.emit("playerMoved", data);  
} catch (JSONException e) {  
 Gdx.*app*.log("SOCKET.IO", "Error sending update data");  
}

Và tại index.js, để lắng nghe sự kiện trên:

socket.on('playerMoved',function(data){  
 //phát tín hiệu cho tất cả các client khác rằng một player đã di chuyển  
 socket.broadcast.emit('onePlayerMoved',data);

});

Và lại tại Android Studio, ta lắng nghe sự kiện này bằng cách:

socket.on("oneplayerMoved", new Emitter.Listener() {  
 @Override  
 public void call(Object... args) {

//Xử lý tùy ý  
 handlePlayerMovedEvent(args);  
 }  
});

Cách thức và thứ tự hoạt động của các công việc trên như sau:

Player của một client nào đó di chuyển và phát ra sự kiên “playerMoved”. Server lắng nghe sự kiện này và thông báo cho tất cả các client khác biết là một player đã di chuyển bằng sự kiện “onePlayerMoved”. Tại một client khác, client này lắng nghe sự kiện “onePlayerMoved” từ server.

Trên đây chỉ là một ví dụ về cách gửi và nhận các thông tin giữa các client và server. Tất nhiên trong game, chúng ta có khá nhiều sự kiện cần phải xử lý như tạo lập room, join room, thoát, … Tất cả các sự kiện này có thể tham khảo trong source code của game.

4. Giao diện toàn bộ chương trình:

IV. Kết luận:

1. Kết luận:

2. Hướng mở rộng:

- Thêm nhiều Map hơn

- In-Game Map Editor: Cho phép người chơi tự tạo lập Map mà người chưa mong muốn.

- Hệ thống tính điểm: Hệ thống so sánh điểm số giữa những người cùng chơi (Ví dụ bạn bè trên Facebook) để tăng tính cạnh tranh và hấp dẫn.

- Mời bạn bè cùng chơi: Cho phép người dùng mời một người bạn vào cùng chơi.