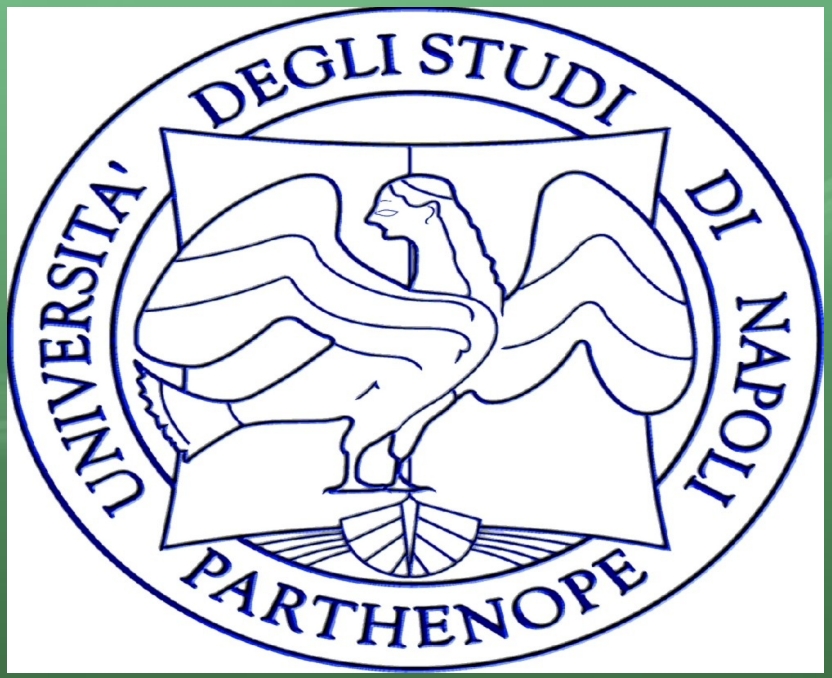
Progetto Programmazione 3 e Lab Programmazione 3

Anno 2014-2015



Michael Nicolella, matr. 0124/45


Platform Game
Rhaegal
The Little Dragon
Start
Commands
Scores
Quit

Introduzione

L’obiettivo di questo progetto è quello di realizzare un gioco di tipo piattaforma 2D con Java, in versione desktop.

Il gioco, che si chiama “Rhaegal”, è composto da 4 livelli principali con difficoltà sempre crescenti, più un livello boss.

Come si può notare dalla diapositiva precedente, appena si avvia il gioco abbiamo a nostra disposizione 4 opzioni:

- Start: avvia il gioco;
- Commands: visualizza i relativi comandi di gioco;
- Scores: visualizza gli ultimi 15 punteggi effettuati giocando;
- Quit: termina il gioco.

Il gioco è dotato di musiche diverse per ogni livello più vari effetti sonori per diverse azioni di gioco. È Inoltre possibile in qualsiasi momento stoppare il gioco tramite il tasto 'Esc' generando uno stato di pausa ed è possibile ritornare alla schermata del menù principale tramite il tasto 'w', perdendo però tutti i progressi di gioco.

L'obiettivo del gioco è quello di, come in tutti i giochi del genere piattaforma, riuscire ad arrivare alla fine del livello con il maggior numero di vite e punteggio.

Oggetto Player

Il Player può essere utilizzato dall'utente tramite la tastiera. Le frecce 'destra' e 'sinistra' muovono il personaggio, 'w' salta, 'e' plana, 'r' attacca e 'f' spara una palla di fuoco. Per fare tutto ciò è implementata l'interfaccia KEYLISTENER.

Implementazione

Il metodo keyPressed() riceve l'evento generato quando viene premuto un tasto.

```
public void keyTyped(KeyEvent key) {}

public void keyPressed(KeyEvent key) {
    Keys.keySet(key.getKeyCode(), true);
}

public void keyReleased(KeyEvent key) {
    Keys.keySet(key.getKeyCode(), false);
}
```

La classe Keys viene utilizzata per aumentare l'astrazione del codice in modo da poter gestire facilmente eventuali modifiche.

```
public static boolean keyState[]    = new boolean[NUM_KEYS];

public static int UP        = 0;
public static int LEFT      = 1;
public static int DOWN      = 2;
public static int RIGHT     = 3;
public static int BUTTON1   = 4;
public static int BUTTON2   = 5;
public static int BUTTON3   = 6;
public static int BUTTON4   = 7;
public static int ENTER     = 8;
public static int ESCAPE    = 9;

public static void keySet(int i, boolean b) {

    if      (i == KeyEvent.VK_UP)    keyState[UP]      = b;
    else if (i == KeyEvent.VK_LEFT)  keyState[LEFT]    = b;
    else if (i == KeyEvent.VK_DOWN)  keyState[DOWN]    = b;
    else if (i == KeyEvent.VK_RIGHT) keyState[RIGHT]   = b;
    else if (i == KeyEvent.VK_W)     keyState[BUTTON1] = b;
    else if (i == KeyEvent.VK_E)     keyState[BUTTON2] = b;
    else if (i == KeyEvent.VK_R)     keyState[BUTTON3] = b;
    else if (i == KeyEvent.VK_F)     keyState[BUTTON4] = b;
    else if (i == KeyEvent.VK_ENTER) keyState[ENTER]   = b;
    else if (i == KeyEvent.VK_ESCAPE)keyState[ESCAPE]  = b;
}
```

Il metodo handleInput() setta l'azione da intraprendere.

```
public void handleInput() {

    player.setLeft(Keys.keyState[Keys.LEFT]);
    player.setRight(Keys.keyState[Keys.RIGHT]);
    player.setJumping(Keys.keyState[Keys.BUTTON1]);
    player.setGliding(Keys.keyState[Keys.BUTTON2]);

    if(Keys.isPressed(Keys.BUTTON3))    player.setScratching();
    if(Keys.isPressed(Keys.BUTTON4))    player.setFiring();
}
```

Caratteristiche Player

Health (Salute): se ne perde 1 collidendo con gli enemies; se ne ottiene 1 raccogliendo un cristallo; arrivato a 0 si perde una vita.

Fireball: numero massimo di attacchi possibili; cambia in base al livello e si ricarica durante il gioco.

Score: è il punteggio di gioco ottenibile raccogliendo cristalli, finendo un livello e sconfiggendo nemici.

Life (Vita): si parte con 3 vite; si perde una vita se si arriva a 0 Health o se si cade nel vuoto; una volta persa una vita si viene rigenerati all'inizio della mappa; si può ottenere 1 vita al termine del livello se si raggiunge un certo punteggio massimo; arrivato a 0 vite...Game Over!

Crystals: sono oggetti di gioco che ti permettono di guadagnare sia punteggio, che varia in base al cristallo ottenuto, sia 1 punto Health.

Enemies: sono i nemici nel gioco, seguono percorsi fissi, cambiando direzione solamente quando collidono con una parte solida della mappa o quando termina il terreno sotto di essi.

Cuore del programma

```
public class GamePanel extends JPanel implements Runnable, KeyListener{

    private Thread thread;
    private boolean running;
    private int FPS = 60;
    private long targetTime = 1000 / FPS;
    private GameStateManager gsm;

    public GamePanel(){
        super();
        setPreferredSize(new Dimension(WIDTH * SCALE, HEIGHT * SCALE));
        setFocusable(true);
        requestFocus();
    }
```

```
    public void addNotify() {
        super.addNotify();
        if(thread == null){
            thread = new Thread(this);
            addKeyListener(this);
            thread.start();}
    }
```

```
    private void init(){
        image = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
        g = (Graphics2D) image.createGraphics();
        running = true;
        gsm = new GameStateManager();
    }
```

```
    public void run(){
        init();

        long start;
        long elapsed;
        long wait;

        while(running){
            start = System.nanoTime();

            update();
            draw();
            drawToScreen();

            elapsed = System.nanoTime() - start;
            wait = targetTime - elapsed / 1000000;
            if (wait < 0) wait = 1;

            try{
                Thread.sleep(wait);}
            catch(Exception e){
                e.printStackTrace();}
        }
    }
```

Implementare l'interfaccia Runnable è un altro modo di creare oggetti Eseguibili, alternativo all'estendere la classe Thread.

Questo metodo è chiamato automaticamente quando l'oggetto JPanel viene aggiunto al componete JFrame; lo rende quindi visibile connettendolo ad una risorsa video nativa.

Init() inizializza l'immagine tramite il metodo createGraphics() di BufferedImage, setta la variabile booleana running al valore true e istanzia un oggetto di tipo GameStateManager per utilizzare i suoi metodi.

Il metodo run() stabilisce un punto di ingresso per un altro thread in esecuzione all'interno del programma. Una volta creato, il nuovo thread viene eseguito dopo aver chiamato il metodo start(), che esegue una chiamata al metodo run(). Questo metodo è il cuore del programma, esso cicla all'infinito in una condizione del tipo while(true). I metodi principali, update(), draw() e drawToScreen() servono rispettivamente per aggiornare tutte le dinamiche di gioco, creare e infine disegnare su schermo l'oggetto grafico finale.

Struttura: update(), draw() e drawToScreen()

I metodi update() e draw() in particolare sono ereditati e, in quasi tutti i casi, sovrascritti dalle varie classi che li implementano.

Il metodo update() è utilizzato dalle varie classi per aggiornare tutto l'ambiente (variabili, oggetti, liste, ecc), come la posizione ed i movimenti del player e le sue caratteristiche, le posizioni e i movimenti degli enemies, il punteggio, il controllo delle collisioni e tutti i calcoli necessari al corretto funzionamento del gioco.

Il metodo draw() è utilizzato per creare oggetti grafici bidimensionali che possano essere in seguito visualizzati a schermo. In pratica ogni classe che implementa questo metodo crea oggetti grafici e stabilisce dove doverli visualizzare a schermo. Tutte le immagini (sprites, backgrounds...) sono mantenute nella cartella /resources e vengono quindi caricate da qui. Le animazioni di gioco vengono create tramite sprites, figure bidimensionali che possono variare posizione rispetto allo sfondo. Ogni sprite contiene più frame che disegnati in sequenza formano l'animazione.

Il metodo drawToScreen() non è mai ereditato, né implementato, da nessun altra classe. Viene chiamato come ultimo metodo ad ogni ciclo del gioco ed è utilizzato per disegnare a schermo l'immagine(frame) del gioco, in quel istante, che le varie classi hanno creato tramite i vari metodi draw(). Infatti ogni volta che una classe invoca il metodo draw() non fa altro che aggiungere (sovrapporre) o eliminare un oggetto grafico alla stessa immagine che si sta creando e che alla fine sarà disegnata su schermo come immagine, che rappresenta il frame.

Esempio di update() per la classe Level_1

```
public void update(){

    handleInput();

    if(player.getHealth() == 0 || player.getY() > tileMap.getHeight()){
        eventDead = blockInput = true;
    }

    if(player.contains(crystal)){
        if(!player.getGreenCrystal()){
            sfx.get("crystal").play();
            crystal.shouldRemove(true);
            player.setScore(1000);
            player.setHealth(player.getHealth() + 1);
            player.setGreenCrystal(true);}
    }

    if(door.contains(player)){eventNextLevel = blockInput = true;}
    if(eventDead) eventDead();
    if(eventNextLevel) eventNextLevel();
    if(eventStart) eventStart();

    player.update();
    player.checkAttack(enemies);
    tileMap.setPosition(GamePanel.WIDTH / 2 - player.getX(),
            GamePanel.HEIGHT / 2 - player.getY());
    bg.setPosition(tileMap.getX(), tileMap.getY());

    for(int i=0; i<enemies.size(); i++){
        Enemy e = enemies.get(i);
        e.update();
        if(e.isDead()){
            player.setScore(e.getScore());
            enemies.remove(i);
            i--;
            explosions.add(new Explosion(e.getX(), e.getY(), levelStateNum));
            sfx.get("explosion").play();}
    }

    for(int i=0; i<explosions.size(); i++){
        explosions.get(i).update();
        if(explosions.get(i).shouldRemove()){
            explosions.remove(i);
            i--;}
    }
    door.update();
    crystal.update();
}
```

Controllo le condizioni per perdere una vita.

Controllo le condizione per ottenere il cristallo verde.

Vari controlli utilizzati per innescare degli eventi, come la perdita di una vita o il superamento del livello.

Effettuo un controllo sugli enemies, li aggiorno e nel caso ne sia stato sconfitto uno, aggiorno il punteggio, lo rimuovo dallo schermo e lo sostituisco con un'esplosione con effetto sonoro.

Controllo se si è verificata un'esplosione e nel caso la gestisco.

Esempio di draw() per la classe Level_1

```
public void draw(Graphics2D g){

    bg.draw(g);
    tileMap.draw(g);
    player.draw(g);
    hud.draw(g);
    door.draw(g);
    crystal.draw(g);

    for(int i=0; i<enemies.size(); i++){
        enemies.get(i).draw(g);
    }

    for(int i=0; i<explosions.size(); i++){
        explosions.get(i).setMapPosition((int)tileMap.getX(), (int)tileMap.getY());
        explosions.get(i).draw(g);
    }

    g.setColor(java.awt.Color.BLACK);
    for(int i = 0; i < tb.size(); i++) {
        g.fill(tb.get(i));
    }
}
```

Disegno vari oggetti di gioco, come background, la tilemap, il player, l'head-up display, la porta per il livello successivo ed il cristallo.

Disegno tutti gli enemies restanti nella lista.

Nel caso ci siano esplosioni setto la loro posizione, presa precedentemente nel metodo update(), e le disegno.

Coloro dei rettangoli neri. La variabile tb è un ArrayList<> del tipo Rectangle, questi vengono utilizzati all'inizio ed alla fine come effetto visivo del tipo transizione tra un livello e l'altro.

Collisioni con parti della mappa

```
public void calculateCorners(double x, double y){
    int leftTile = (int)(x - cwidth / 2) / tileSize;
    int rightTile = (int)(x + cwidth / 2 - 1) / tileSize;
    int topTile = (int)(y - cheight / 2) / tileSize;
    int bottomTile = (int)(y + cheight / 2 - 1) / tileSize;
    if(topTile < 0 || bottomTile >= tileMap.getNumRows() ||
            leftTile < 0 || rightTile >= tileMap.getNumCols()){
        topLeft = topRight = bottomLeft = bottomRight = false;
        return;
    }

    int tl = tileMap.getType(topTile, leftTile);
    int tr = tileMap.getType(topTile, rightTile);
    int bl = tileMap.getType(bottomTile, leftTile);
    int br = tileMap.getType(bottomTile, rightTile);

    topLeft     = tl == Tile.BLOCKED;
    topRight    = tr == Tile.BLOCKED;
    bottomLeft  = bl == Tile.BLOCKED;
    bottomRight = br == Tile.BLOCKED;
}
```

Le “tile”, cioè le piastrelle di gioco con cui è composta la mappa, sono di due tipi: le BLOCKED sono quelle con le quali si collide e le NORMAL le restanti. Il metodo calculateCorners() calcola gli angoli dell'oggetto, che è un quadrato, nella mappa e poi tramite il metodo getType() della classe TileMap, che prende in input una riga ed una colonna della mappa, otteniamo il tipo di tile. A questo punto tramite delle espressioni condizionali modifichiamo delle variabili booleane rappresentanti i quattro angoli controllando se questi ultimi si trovano nelle coordinate di una “blocked tile” per poter in seguito controllare eventuali collisioni con parti solide della mappa.

Collisioni con parti della mappa

```
public void checkTileMapCollision(){
    currCol = (int)x / tileSize;
    currRow = (int)y / tileSize;
    xdest = x + dx;
    ydest = y + dy;
    xtemp = x;
    ytemp = y;

    calculateCorners(x, ydest);
    if(dy < 0){
        if(topLeft || topRight){
            dy = 0;
            ytemp = currRow * tileSize + cheight / 2;}
        else{
            ytemp += dy;}
    }
    if (dy > 0){
        if(bottomLeft || bottomRight){
            dy = 0;
            falling = false;
            ytemp = (currRow + 1) * tileSize - cheight / 2;}
        else{
            ytemp += dy;}
    }

    calculateCorners(xdest, y);
    if(dx < 0){
        if(topLeft || bottomLeft){
            dx = 0;
            xtemp = currCol * tileSize + cwidth / 2;}
        else{
            xtemp += dx;}
    }
    if (dx > 0){
        if(topRight || bottomRight){
            dx = 0;
            xtemp = (currCol + 1) * tileSize - cwidth / 2;}
        else{
            xtemp += dx;}
    }

    if(!falling){
        calculateCorners(x, ydest + 1);
        if(!bottomLeft && !bottomRight){
            falling = true;}
    }
}
```

curCol e curRow si riferiscono alla colonna e riga corrente, x e y sono le coordinate, dx e dy indicano le direzioni e xdest e ydest sone le destinazioni finali dell'oggetto. Dopo aver calcolato la posizione dei 4 angoli dell'oggetto, tramite il metodo calculateCorners(), controlliamo se qualcuno di essi è andato in collisione con una “tile” del tipo BLOCKED. Il controllo avviene sia sull'asse delle ordinate che su quello delle ascisse per controllare se siamo bloccati da una tile dal basso, dall'alto, da destra o da sinistra. Nel caso in cui avvenga una collisione utilizzo delle variabili temporanee xtemp e ytemp per far tornare l'oggetto nella sua posizione originaria. Come ultima cosa controllo se gli angoli in basso a destra ed in basso a sinistra collidono con un oggetto della mappa, in caso contrario setto la variabile booleana falling a true cosi da innescare il processo di caduta dell'oggetto verso il basso, in modo da simulare la gravità.

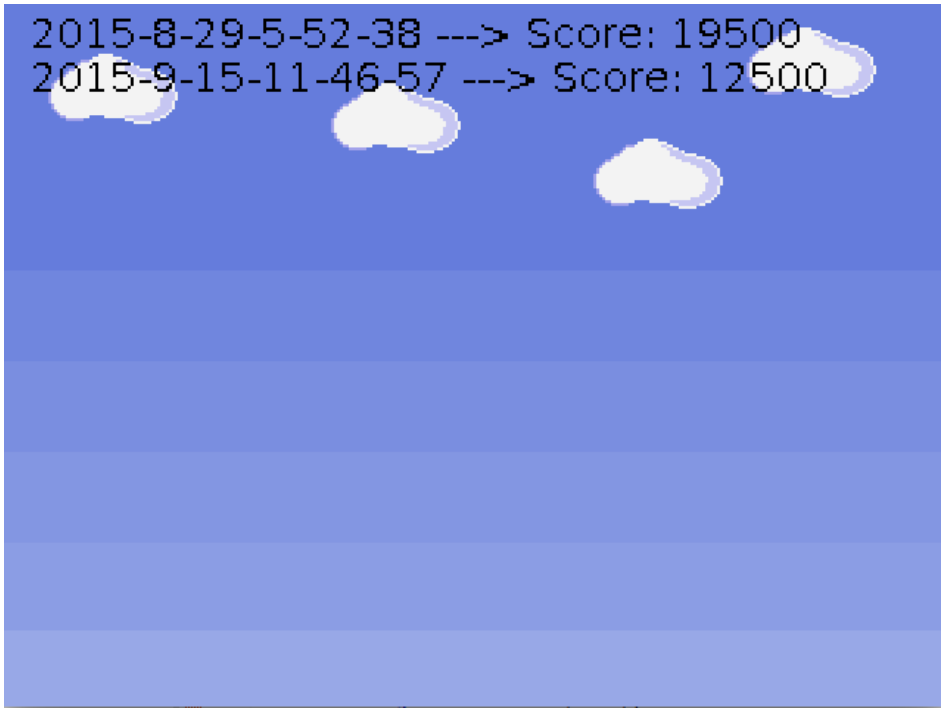
Collisioni e attacchi con altri oggetti della mappa

```
public void checkAttack(ArrayList<Enemy> enemies){

    for(int i=0; i<enemies.size(); i++){
        Enemy e = enemies.get(i);

        if(scratching){
            if(e.getFlinching())
                return;
            if(e.getX() > x && e.getX() < x + scratchRange
                    && e.getY() > y - height / 2 && e.getY() < y + height / 2){
                e.hit(scratchDamage);
                e.setFlinching(true);}
        }

        for(int j=0; j<fireBalls.size(); j++){
            if(fireBalls.get(j).intersects(e)){
                if(e.getFlinching())
                    return;
                e.hit(fireBallDamage);
                fireBalls.get(j).setHit();
                break;}
        }

        if(intersects(e))
            hit(e.getDamage());
    }
}

public void hit(int damage){
    if(flinching) return;
    health -= damage;
    if(health < 0) health = 0;
    if(health == 0) dead = true;
    flinching = true;
    flinchTimer = System.nanoTime();
}
```

Il metodo checkEnemyAttack() effettua un controllo per tutti e due i tipi di attacchi possibili. Prima controlla l'attacco scratching e nel caso il nemico sia già stato colpito da poco e si trova quindi in uno stato di flinching (cioè lampeggia per un breve periodo di tempo) non succede niente. Nel caso contrario, controlla se l'enemy si trova nel raggio dell'attacco del player ed in caso affermativo subisce il danno ed entra nello stato di flinching. Il secondo controllo riguarda la palla di fuoco, nel momento in cui ne viene creata una la si aggiunge ad un contenitore, quindi controllo che ce ne sia almeno una tramite il for e, in caso affermativo, controllo se avviene una collisione con un enemy tramite il metodo intersects() della classe Rectangle2D. Infine, sempre tramite intersects(), controllo se il player è stato colpito da un enemy, in caso affermativo chiamo il metodo hit() che aggiorna lo stato del giocatore aggiungendo il danno appena subito.

Alcuni screenshot...scontro con il Boss

Alcuni screenshot...schermate commands e score



Il punteggio viene accumulato durante tutto il gioco e nel momento in cui si sconfigge il boss finale viene salvato in coda ad un file, generato automaticamente in caso non esistesse, con la data corrente. Nel momento in cui lo si va a controllare tramite l'apposita opzione nel menù principale, il programma va a leggere gli ultimi 15 punteggi dal file e li visualizza a schermo nel modo in cui si evince dallo screenshot a destra.

Sviluppi futuri...

- Creazione di un database per sostituire il file dei salvataggi creando una classifica dei risultati;
- Aggiunta di nuovi livelli di gioco con tilemap differenti;
- Aggiunta di nuovi enemies e nuovi Boss con cui combattere;
- Sviluppo del gioco su piattaforma Android.

Risorse

Tutto il materiale (risorse varie, backgrounds, tileset, musica, effetti sonori, sprites) è free ed opensource con licenza creative commons ed è stato preso dal sito: ***opengameart.org***