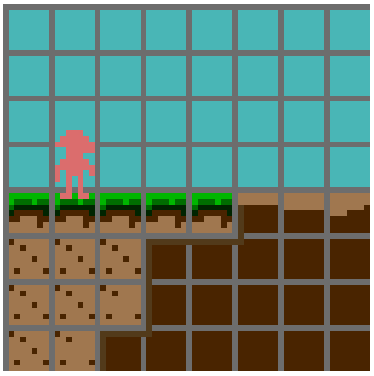
Frédéric Delorme<frederic.delorme@snapgames.fr>



GDJ102 // Game Development Java

Game Loop pattern et user interaction

http://github.com/snapgames/GDJ102.git

GameDev Java Basics on google classroom

Context

Ce que vous allez apprendre

Vous allez découvrir ce qu'est le pattern de la boucle de jeu (Game Loop in english). C'est ici que tout se passe dans un jeu.

Ce que vous pourrez faire

Afficher des choses à l'écran, détecter des interactions utilisateur via des évènements clavier et animer votre jeu.

Pré-requis

Afin de bien cadrer le débat, il est indispensable que l'élève connaisse

- la programmation en langage JAVA,
- maîtrise une IDE de son choix: Eclipse, NetBeans ou IntelliJ. *Le cours se déroulera sur Eclipse.*
- les bases de maven,
- les librairies graphiques offerte par la plateforme Java, à savoir AWT, Swing ou même SWT de la fondation Eclipse.
- Il est également possible de passer par le langage JavaFX.
- Être passé par le cours GDJ101 - création d'une fenêtre.

Le projet

Le projet de base sera un projet standard Maven. Sa structure sera la suivante :

```
game-102/
  |_ src
  |  |_ main
  |  |  |_ java
  |  |  |_ resources
  |  |_ test
  |  |  |_ java
  |  |  |_ resources
  |_ pom.xml
  |_ README.md
  |_.gitignore
```

Il est hébergé sur un répertoire github.com : https://github.com/SnapGames/GDJ102

Le point d'entrée est le fichier `pom.xml` décrivant à la commande maven comment construire le projet. Pour en découvrir d'avantage sur maven, je vous invite à visiter les posts sur ce blog :

1. https://mcgivrer.wordpress.com/2014/04/08/maven-premier-pas-pour-les-non-sachants/
2. https://mcgivrer.wordpress.com/2014/04/18/maven-step-2-dans-le-gras-du-sujet/

En attendant, sachez qu'il faudra exécuter la commande ci-dessous pour compiler:

```
$> mvn clean install↲
```

Et pour lancer l'exécution de notre programme :

```
$> mvn exec:java↲
```

La magie opérera via des plugins spécifiques à Maven. nous découvrirons ceux-ci ultérieurement, quand il s'agira de packager notre jeu pour le livrer.

Les fichiers `README.md` , `.gitignore` et .travis.yml sont présents simplement pour le côté pratique :

1. le premier permet de décrire le contenu de notre projet ainsi que comment le compiler et l'exécuter ,
2. le deuxième est un fichier destiné au gestionnaire de source git, pour lui indiquer quels sont les fichiers exclus de la gestion de version. Je vous en laisse découvrir le contenu par vous-même.
3. le troisième permet, via les plateformes github.com et travisci.org de procéder à du build automatique sur le repository git, afin de s'assurer qu'il n'y a aucune erreur lorsqu'on pousse du nouveau code sur un projet existant (Guide des outils

et bonnes pratiques à venir).

GameLoop'ing !

Dans tout jeu vidéo, il y a une boucle principale. Enfin, dans tous les vieux jeux vidéo ou encore appelé Jeux vidéo Old School. Aujourd'hui les frameworks aidant à l'écriture de jeux sont tellement complexe, qu'il nécessitent plusieurs threads en parallèle pour pouvoir exécuter l'ensemble des services d'un jeu.

Unity, Unreal sont des noms fameux des jeux AAA.

Ici nous n'allons pas du tout parler d'eux :)

La boucle

Dans notre jeu, nous allons partir du concept simple de la boucle infinie:

```
while(!exit){
  input();
  update();
  render();
}
```

Si on interprète ce code, on voit que tant qu'on n'a pas explicitement demandé de sortir du jeu (!exit), on exécute les méthodes input(), update() et render();

- La méthode **input()** sera chargé de la détection des actions effectuée par l'utilisateur et des inputs qu'il envoie au jeu,
- **update(**) permettra, en fonction des entrées utilisateur et des mécaniques interne du jeu en place de faire évoluer les objets manipulés; fair avancer le joueur, faire réagir les ennemis, animer des objets, etc… ,
- **rende()** permettra l'affichage à proprement parler des objets graphiques du jeu.

Il est aussi indispensable d'avoir une méthode pour l'initialisation du jeu **initialize()**, ainsi qu'une méthode pour encapsuler cette boucle: **loop()**.

Enfin, afin de rendre une zone de jeu propre, il est également nécessaire de prévoir une méthode de libération des ressources: **release()**;

Enfin, il nous faut un point d'entrée pour lancer notre jeu, je propose une méthode **run()**.

Maintenant que nous savons tout cela, nous pouvons tenter une implémentation de la chose .

Reprenons notre classe Game, et ajoutons lui le nécessaire:

```
class Game extends JPanel {
```

On trouve quelques attributs

```
    private String title=”game”;
    private Dimension dimension = null;
```

Dont quelques nouveautées:

```
    private boolean exit=false;
    private Graphics2D g;
    private InputHandler inputHandler;
    private Window window;
```

On trouve notre constructeur

```
    public Game(String title){
        this.title=title;
        this.dimension = new Dimension(640,400);
    }
```

puis quelques nouvelles méthodes:

```
    // Initialize the game.
    public void initialize(){
    }
    // manage all user/player input
    public void input(){
    }
    // update all game’s entities
    public void update(){
    }
    // render all graphical objects.
    public void render(Graphics2D g){
    }
    // free all resources
    public void release(){
    }
```

La fameuse boucle principale

```
// main game loop
public void loop(){
  while(!exit){
      input();
      update();
      render(g);
  }
}
```

Le point d'entrée de l'exécution du jeu

```
// Start the game
public void run(){
  initialize();
  loop();
  release();
}
```

et les inénarrables getters/setters

```
// retrieve the title of the game
public void getTitle(){
      return title;
}
}
```

Si l'on souhaite lancer l'exécution de notre jeu, nous aurons maintenant :

```
public void main(String[] argv){
  Game game = new Game("MyGame");
  new Window(game);
  game.run();
}
```

Si vous lancez l'exécution , vous ne constaterez rien de bien folichon à observer.
Aussi, nous allons devoir ajouter un peu d'interactivité.

Window

La classe Window doit être quelques peu modifiée pour permettre la déclaration du **InputHandler** comme key listener de celle-ci.

Dans le constructeur de **Window(Game)** il faut ajouter la déclaration du *KeyListener* et il faut également déclarer la fenêtre comme fenêtre active dans la classe game.

```
public Window(Game game) {
            game.setSize(game.getDimension());
            frame = new JFrame(game.getTitle());
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setContentPane(game);
            frame.setLayout(new BorderLayout());
            frame.setSize(game.getDimension());
            frame.setPreferredSize(game.getDimension());
            frame.setMaximumSize(game.getDimension());
            frame.setResizable(false);
```

On déclare le KeyListener:

```
            // add the Game InputHandler as a KeyListener
            frame.addKeyListener(game.getInputHandler());

            frame.pack();
            frame.setVisible(true);
```

On positionne cette fenêtre comme celle active pour le jeu.

```
            // set this window as the current active one in the Game.
            game.setWindow(this);
      }
```

Enfin , dans la méthode release() de la classe Game, il faut libérer la fenêtre créée:

```
      private void release() {
            window.frame.dispose();
      }
```

Et fermer proprement notre application avec un appel à *System.exit(0)* en fin de méthode **run()**:

```
      public void run() {
            initialize();
            loop();
            release();
            System.exit(0);
      }
```

Nous pouvons maintenant passer à l'implémentation dans le détail du *KeyListener* dans la classe **InputHandler**.

l'InputHandler

La classe que nous allons appeler InputHandler sera une implémentation des interfaces standard Java que sont KeyListener et MouseListener.

```
public class InputHandler implements KeyListener, MouseListener {
...
}
```

Si nous regardons de plus prêt, nous verrons que ces interface imposent la création des méthodes suivantes:

1. pour la gestion des touches (*KeyListener*):

```
public void keyPressed(KeyEvent k){
  ...
}
public void keyReleased(KeyEvent k){
  ...
}
public void keyTyped(KeyEvent k){
  ...
}
```

2. la gestion de la souris (*MouseListener*) se fera dans un deuxième temps.

Nous allons garder en mémoire un cache des touches appuyées et des touches relâchées, et à chaque passage dans la boucle (**loop()**) nous écouterons ces caches, et nous les remettrons à zéro.

nous allons gérer au maximum 256 touches, déclarons ces deux tableaux de booléen pour conserver ces états :

```
boolean[] keyPressed = new boolean[256];
boolean[] keyReleased = new boolean[256];
```

Le premier servira à tracer les touches appuyées (keyPressed), le second les touches relâchées (keyReleased).

Pour interroger le handler , nous ajouterons 2 méthodes , l'une pour vérifier l'appui d'une touche, l'autre son relâchement:

```
public boolean getKeyPressed(int k) {
```

```
        assert (k >= 0 && k < 256);
        return keyPressed[k];
    }


    public boolean getKeyReleased(int k) {
        assert (k >= 0 && k < 256);
        return keyReleased[k];
    }
```

Note
Il est important de noter que la valeur passée en paramètre et permettant de vérifier une touche est vérifiée vis-à-vis de la taille du buffer d'état de touche.

Ainsi, si nous revenons à la méthode **Game#input()** nous pouvons maintenant vérifier le code d'une touche:

```
    private void input() {
        // Process keys
        if (inputHandler.getKeyReleased(KeyEvent.VK_ESCAPE)) {
            setExit(true);
        }
        inputHandler.clean();
    }
```

Ici nous testons la touche ESCAPE. si celle-ci a été relâchée, nous demandons à sortir du jeu par un appel à **Game#setExit()**.

Note
*on peut voir ici que les caches des états des touches keyPressed[] et keyReleased[] sont remis à zéro en fin de vérification avec l'appel à la méthode **InputHandler#clean()**.*

Nous lançons maintenant notre jeu modifié via la commande ci-dessous:

```
$> java -jar gdj102-0.0.1-SNAPSHOT.jar
```

la fenêtre apparaît maintenant:

un appui sur une touche provoque des sorties sur la console, indiquant les événements détectés:

```
User pressed the '18' key
User released the '18' key
User pressed the '18' key
User released the '154' key
User released the '18' key
```

La fenêtre se ferme dès l'appui sur la touche ESCAPE.

GameLoop : le retour

Nous avons réalisé une première version de notre boucle de jeu, mais si vous regarder la consommation CPU, elle atteint des sommets, et pour cause, la boucle est infinie.

Si nous voulons donner une chance au CPU de respirer un peu, il est nécessaire de faire attendre notre programme quelques millisecondes.

l'algorithme de la boucle devient alors:

```
while(!exit){
        input();
        update();
        render();
        wait(delay);
}
```

On ajoute donc une méthode **Game#wait(int)**

```
public void wait(int delay){
```

```
        try{
          Thread.sleep(delay);
        }catch(Exception e){
          System.out.err("Unable to wait");
        }
      }
```

Nous avons maintenant une jolie boucle de jeu. ajoutons une balle, car c'est toujours mieux et plus visuel avec un balle !

Reprenons notre object Game et ajoutons lui quelques attributs afin de gérer l'affichage une balle bondissante de côté en côté.

```
class Game … {

  ...
  // ball position
  int x=0,y=0;
  // ball velocity
  int dx=0,dy=0;
  // ball diameter
  int r=16;
  ...
  /**
   * Update game internals
   */
  private void update() {
    x += dx;
    y += dy;
    if (x <= 0 || x >= getWidth() - 16) {
      dx = -dx;
      color = Color.YELLOW;
    }
    if (y <= 0 || y >= getHeight() - 16) {
      dy = -dy;
      color = Color.YELLOW;
    }
  }

  /**
   * render all the beautiful things.
   *
   * @param g
   */
  private void render(Graphics2D g) {

    g.setColor(color);
    g.fillArc(x, y, r, r, 0, 360);
    color = Color.RED;
  }
}
```

Si vous relancer la classe Game, vous verrez une jolie balle rouge rebondir de part en part de la fenêtre.

Regardons de plus près la méthode **update()**

```
  x += dx;
  y += dy;
```

A chaque appel de la méthode, la position (x,y) de la balle est mise à jour en fonction des déplacements axiaux de celle-ci qui sont notés (dx,dy).

Ensuite, on s'assure que la balle ne part pas dans l'espace mais reste bien dans les limites de l'écran :

```
if (x <= 0 || x >= getWidth() - 16) {
  dx = -dx;
  color = Color.YELLOW;
}
if (y <= 0 || y >= getHeight() - 16) {
  dy = -dy;
  color = Color.YELLOW;
}
```

Ainsi les déplacement de la balle seront limités à la surface de la fenêtre.

Vous pourrez noter que nous n'arrêtons pas la balle mais changeons la direction de déplacement sur l'axe concerné.

Dans la méthode de rendu du jeux, nous traçons la balle en utilisant la fonction **Graphics2D#drawArc()**:

```
g.setColor(color);
g.fillArc(x, y, r, r, 0, 360);
color = Color.RED;
```

Le paramètre semblent évident, il s'agit de la position x,y du rayon sur les axes **x** et **y** (ici la valeur sera **r**) et la taille de l'arc ira de **0°** à **360°**.

Note
Une petite astuce est à noté, pour provoquer un léger flash lors de la collision de la balle contre un bord, nous changeons la couleur d'affichage dans la méthode update, puis la méthode render est appelée pour le rendu, où juste après nous remettons à sa couleur initiale la balle.

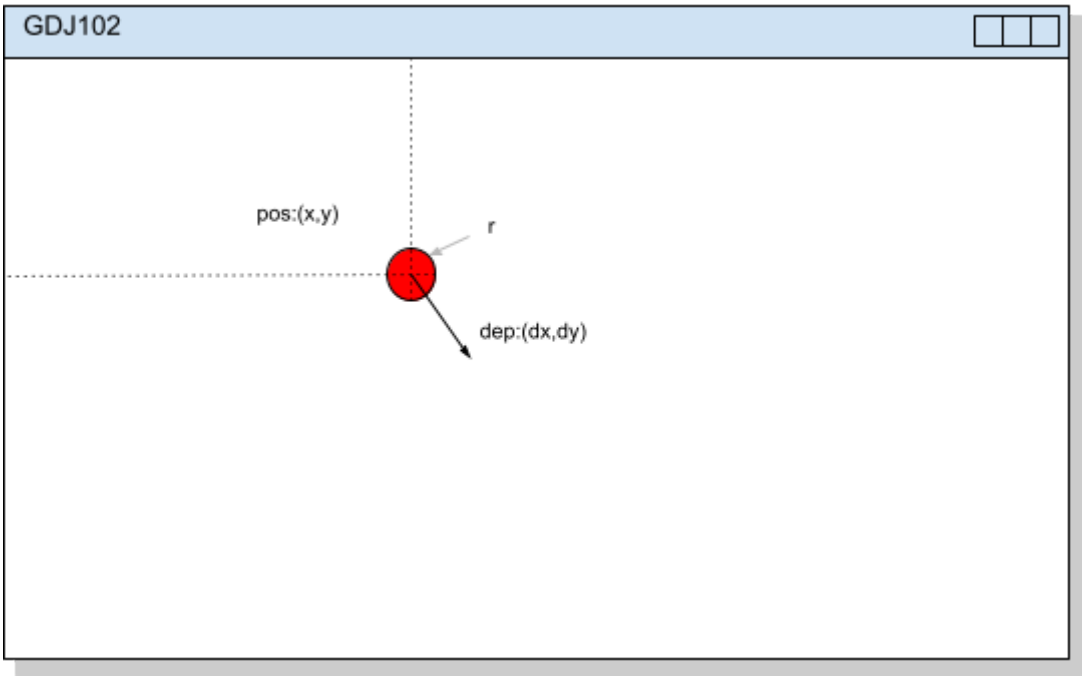


*figure 1 - la balle rouge de rayon **r** à la position (**x,y**) avec un vecteur déplacement (**dx,dy**)*

Si vous lancez maintenant l'exécution de la démo **gdj102**, vous obtiendrez à l'écran la fenêtre ci-dessous:

figure 2 - un timide affichage d'une balle rouge et de la fréquence de rendu.

Note
Vous remarquerez sans doute la présence d'une nouvelle Class qui n'est ici qu'à titre pratique de réalisation des captures d'image du rendu : ImageUtils.
*la méthode mise en oeuvre ici est **ImageUtils#screenshot(BufferedImage)**. en pressant S une capture du rendu est effectué et stocké à la racine du répertoire utilisateur (la variable system 'user.home' en java).*

Conclusion

Nous en avons fini pour ce cours **gdj102**.

Vous avez découvert:

- les bases de la boucle de jeux,
- intercepter les actions issues des touches du clavier,
- affiché une baballes, et rouge qui plus est !