

Introduction à la création de jeux-vidéo avec LibGDX

11 mars 2018

1 Introduction

1.1 Problématique de la création de jeux-vidéo

La création de jeux vidéo implique la croisée de multiples talents bien différents comme la création de scénario, de ressources graphiques et sonores et la programmation du moteur de jeu par exemple. C'est pour cette raison que la création de jeux-vidéos est longtemps restée inaccessible aux non-professionnels. Néanmoins, ces dix dernières années, la création de jeux-vidéos est devenue de plus en plus accessible aux amateurs passionnés grâce à la sortie d'outils de développement gratuit qui automatisent certaines étapes rébarbatives ou trop bas niveau qui causaient la fin de beaucoup de projets. Les grands du milieu tels qu' Epic Game ou Unity ont bien compris cet engouement pour la création de jeux et ont décidé de publier une version gratuite très peu limitée de leurs moteurs de jeux professionnels(respectivement Unreal Engine 4¹ et Unity²).

Ces moteurs proposent de nombreux outils graphiques qui permettent d'accélérer le travail des artistes, le prototypage et une gestion de la technique de fond (l'affichage par exemple) qui permet aux développeurs de ne programmer que l'essentiel i.e. les mécaniques de jeu (couramment appelées gameplay). Cependant, pour une première création de jeu, il peut être intéressant de ne pas utiliser ces outils afin de comprendre comment ils fonctionnent. A l'inverse, il est peu conseillé d'utiliser des bibliothèques bas niveau si le but est produire un "vrai" jeu vidéo. Les framework de création de jeux-vidéos "100% code" se situent entre ces deux niveaux et sont donc très intéressants.

LibGDX fait parti des frameworks de développement et proposent de nombreuses fonctionnalités et avantages ; LibGDX est un framework :

1. Unreal Engine 4, moteur mondialement utilisé dans l'industrie <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>

2. Unity moteur de jeu très populaire chez les professionnels indépendants <https://unity3d.com/fr>

- multi-plaforme : on écrit le code une fois et LibGDX permet d’exporter le jeu sous de nombreuses plateformes telles que Windows, Mac, Linux, Android ou IOS,
- open source,
- qui possède de nombreuses extensions,
- qui implémente de nombreuses fonctions facilitant l’affichage.

1.2 De quoi se compose un jeu-video ?

Le monde de la création vidéo ludique est très imprégné par la culture anglo-saxonne de telle sorte que beaucoup de termes utilisés dans ce milieu sont en anglais ; c’est pourquoi, après avoir introduit les concepts, nous utiliserons directement les mots dans la langue de Shakespeare. Tout d’abord, les ressources graphiques ; on peut distinguer deux sous types. Premièrement, il y a les images statiques qui peuvent servir à faire le fond d’un menu ou représenter un personnage qui ne bouge pas ; dans LibGDX, cela correspond à une instance de la classe *Texture*. Deuxièmement, il a les ressources graphiques appelées sprite qui sont destinées à contenir des données telles que leurs positions en plus d’une image ; elles servent souvent à représenter des personnages et correspondent à une instance de *Sprite* dans LibGDX.

2 Fonctionnement d’une application basique avec LibGDX

2.1 Organisation d’un projet LibGDX

Il est vivement conseillé par la documentation de mettre en place son projet LibGDX avec le gestionnaire de dépendance Gradle par le biais du *LibGdx Project Generator* :



Figure 1 : LibGdx Project Generator

On peut décider d'inclure les extensions que l'on souhaite et choisir les plateformes supportées par notre jeu vidéo. Pour chaque plateforme choisie, l'utilitaire va générer un nouveau projet en dehors du projet principale *core*. Voici donc à quoi ressemble l'architecture d'un projet entier s'appelant Bac A Sable et ayant pour objectif de créer une application seulement à destination des ordinateurs Windows, Mac et Linux.

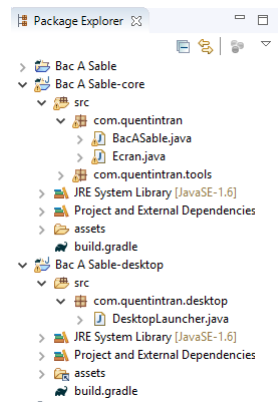


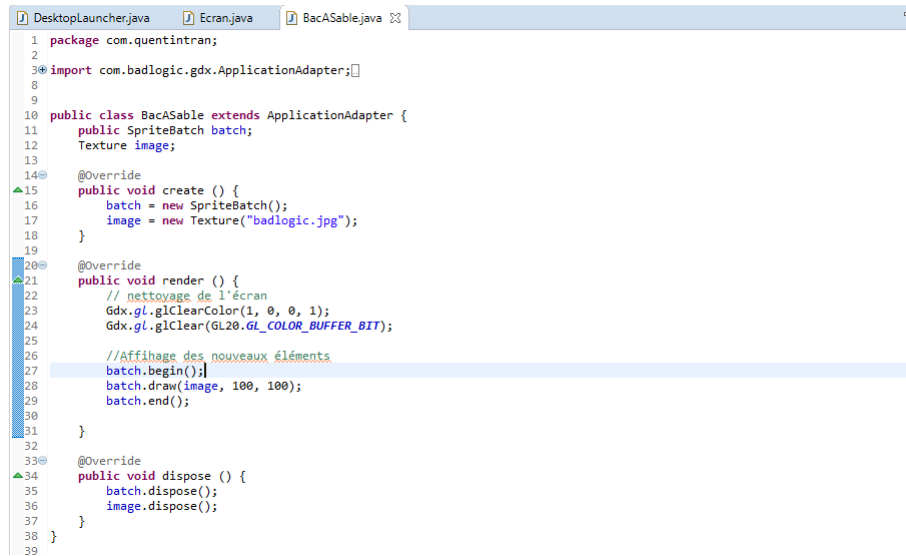
Figure 2 : Hiérarchie d'un projet LibGDX

Le projet Bac A Sable-core contient le code et les ressources communs à tous les supports visés lors du déploiement de l'application. C'est ici que sera écrit 99% du code. Le projet Bac A Sable-desktop contient une classe *DesktopLauncher.java* qui est la classe à exécuter pour tester le jeu sur pc. Cette classe

utilise la classe *BacASable* du projet Bac A Sable-core dont le fonctionnement est détaillé paragraphe suivant.

2.2 Cycle de vie d'une application

Voici le contenu de la classe BacASable du projet cœur, elle se contente ici d'afficher une image située dans le dossier asset du projet.



```
1 package com.quentintran;
2
3 import com.badlogic.gdx.ApplicationAdapter;
4
5
6
7
8
9
10 public class BacASable extends ApplicationAdapter {
11     public SpriteBatch batch;
12     Texture image;
13
14     @Override
15     public void create () {
16         batch = new SpriteBatch();
17         image = new Texture("badlogic.jpg");
18     }
19
20     @Override
21     public void render () {
22         // nettoyage de l'écran
23         Gdx.gl.glClearColor(1, 0, 0, 1);
24         Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
25
26         //Affichage des nouveaux éléments
27         batch.begin();
28         batch.draw(image, 100, 100);
29         batch.end();
30
31     }
32
33     @Override
34     public void dispose () {
35         batch.dispose();
36         image.dispose();
37     }
38 }
39
```

Figure 3 : Classe de base affichant une image

On y trouve deux variables, la première est essentielle à tout projet et est une référence vers une instance de la classe *SpriteBatch*. Cet objet ne doit être instancié qu'une fois dans tout le projet et permet de dessiner sur l'écran. La deuxième variable est la texture que nous souhaitons afficher. La méthode *create* est la première méthode appelée lors du démarrage du jeu, elle fait office de constructeur ; c'est donc ici que nous initialisons les deux champs. La méthode *dispose* est la dernière appelée par le jeu lorsqu'il se ferme, elle permet de libérer la mémoire utilisée par les différentes ressources.

Intéressons-nous maintenant à la méthode *render* qui est la méthode cœur. Un jeu vidéo est comme un film, c'est une succession d'images qui donne l'illusion du mouvement. Tandis qu'au cinéma, on se contente généralement de 24 images/seconde (ou 24 fps pour frame per second), dans un jeu-vidéo les standards sont plutôt de 30, 60 ou 144 fps selon la puissance des ordinateurs. Ce nombre d'images par seconde correspond aux nombres de fois que l'ordinateur a été capable d'appeler la méthode *render* en une seconde : c'est donc elle qui génère chaque image. Pour faire cela, on y trouve d'abord deux fonctions qui nettoient l'écran en supprimant ce qui a été affiché à l'image précédente. Ensuite,

on indique la volonté de vouloir dessiner avec la méthode `batch.begin()` puis on dessine notre image avec le `SpriteBatch` en la plaçant aux coordonnées (100, 100). Enfin on spécifie à l'application qu'on a fini de dessiner avec `batch.end()`. Cette boucle se répète tant que le jeu tourne. Ce cycle de fonctionnement est très bien résumé sur le wiki officiel du framework :

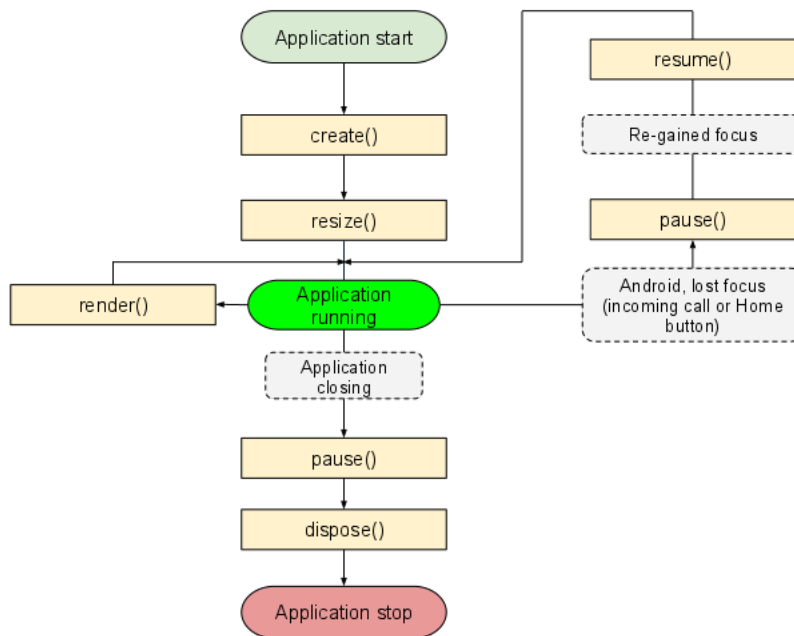


Figure 4 : Cycle de fonctionnement

<https://raw.githubusercontent.com/wiki/libgdx/libgdx/images/70efff32-dd28-11e3-9fc4-1eb57143aee6.png>

La méthode `resume` correspond au moment où le jeu est en pause tandis que la méthode `resize` permet de gérer les redimensionnements de la fenêtre de jeu. Cependant, on ne peut écrire toute notre application dans cette classe `BacASable`. Dès lors que l'on souhaite faire un peu plus que le seul affichage d'une image, le code devient relativement peu clair.

3 Création d'une application complexe avec LibGDX

3.1 La Classe Game et l'interface Screen

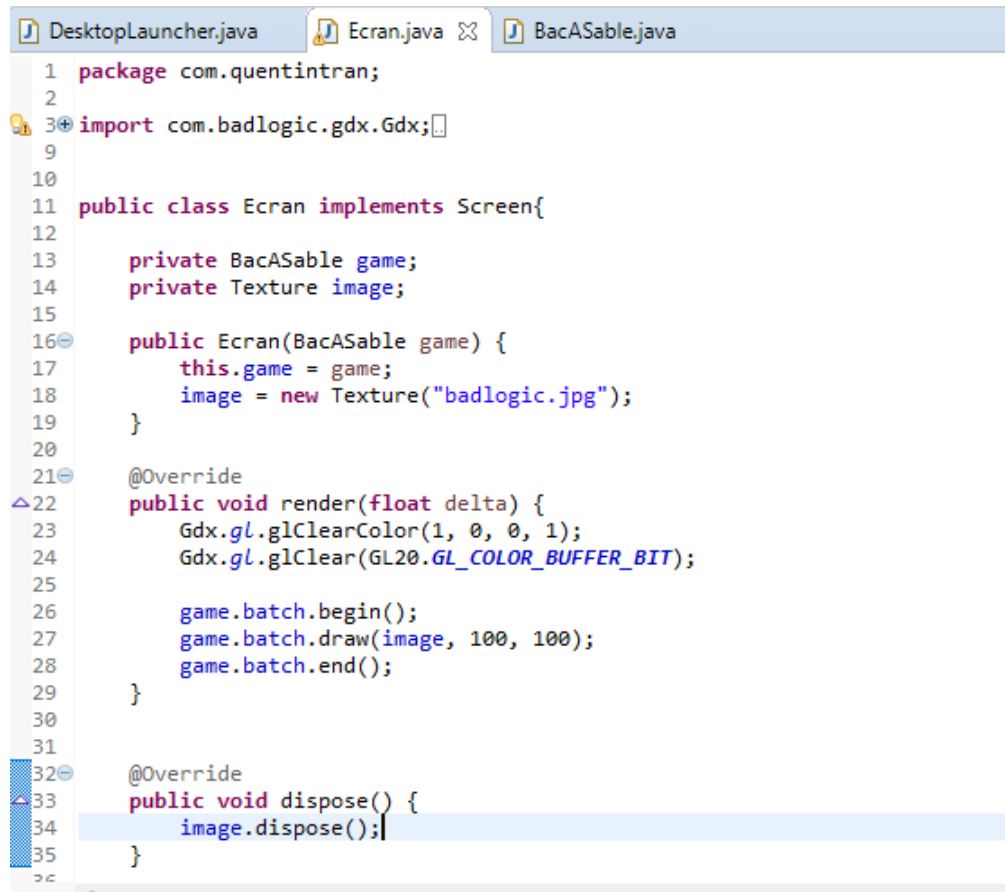
L'architecture présentée en partie 2 pose beaucoup de problèmes quand on souhaite réaliser une application avec plusieurs niveaux, une page d'accueil, ect. Pour combler ce problème, LibGDX nous propose une classe disposant de plus de fonctionnalités que `ApplicationAdapter`, la classe dont héritait notre classe cœur

BacASable.java : c'est la classe *Game*. Elle permet d'organiser l'application en plusieurs "fenêtres" correspondant à des instances différentes de classes mettant en ouvre l'interface *Screen*. On aura par exemple une fenêtre pour l'écran de base, puis une autre pour le premier niveau de notre jeu, etc. Puisque qu'un exemple est plus parlant que mille mots, reproduisons notre code précédent avec cette nouvelle architecture.

The image shows a screenshot of a Java IDE with three tabs: DesktopLauncher.java, Ecran.java, and *BacASable.java. The *BacASable.java tab is active, displaying the following code:

```
1 package com.quentintran;
2
3 import com.badlogic.gdx.Game;
4
5
6
7 public class BacASable extends Game {
8     public SpriteBatch batch;
9
10    @Override
11    public void create () {
12        batch = new SpriteBatch();
13        setScreen(new Ecran(this));
14    }
15
16    @Override
17    public void render () {
18        super.render();
19    }
20
21    @Override
22    public void dispose () {
23        batch.dispose();
24    }
25 }
26
```

Figure 5 : Classe *BacASable*



```

1 package com.quentintran;
2
3 import com.badlogic.gdx.Gdx;
4
5
6
7
8
9
10
11 public class Ecran implements Screen{
12
13     private BacASable game;
14     private Texture image;
15
16     public Ecran(BacASable game) {
17         this.game = game;
18         image = new Texture("badlogic.jpg");
19     }
20
21     @Override
22     public void render(float delta) {
23         Gdx.gl.glClearColor(1, 0, 0, 1);
24         Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
25
26         game.batch.begin();
27         game.batch.draw(image, 100, 100);
28         game.batch.end();
29     }
30
31
32     @Override
33     public void dispose() {
34         image.dispose();
35     }
36
37 }

```

Figure 6 : Notre classe mettant en oeuvre Screen

La classe *BacASable* hérite maintenant de *Game* et appelle la méthode *setScreen* au démarrage avec comme paramètre le Screen *Ecran*. C'est cette classe qui ici va contenir toutes les informations. Le fait de mettre en oeuvre *Screen* pour *Ecran* implique d'implémenter plus de méthodes que celles vues précédemment (i.e. *dispose* et *render*). Cependant, les autres méthodes ne sont pas fondamentales et ne seront donc pas abordées ici. On remarque aussi qu'il n'y a plus de méthode *create*, il faut donc initialiser les champs dans un constructeur. Ce dernier reçoit en argument la classe précédente afin de pouvoir utiliser le *SpriteBatch* pour pouvoir dessiner. Le code reste très proche du premier exemple mais est néanmoins beaucoup mieux organisé. Ainsi, si on veut aller dans un menu d'accueil que l'on aurait créé dans une classe *MenuPrincipale.java* il suffit d'appeler la méthode *game.setScreen(game)* quelque part dans la classe *Ecran*.

3.2 La notion de deltaTime

Nous ne pouvons nous empêcher de remarquer que dans le dernier exemple, la méthode `render` prend en argument un réel `delta`. Ce réel est très important et correspond au temps écoulé entre la génération de l'image actuelle et l'image précédente; cette valeur est très utile pour pouvoir réaliser des actions dans un laps de temps constant. Imaginons que nous souhaitons déplacer l'image que nous avons affiché précédemment le long de l'axe des abscisses. Nous pouvons par exemple écrire ce code ci dans la méthode *render* :

```
batch.draw(image, positionX, 100);  
positionX ++;
```


Ainsi, à chaque image générée, l'image se déplacera d'un pixel à droite. Cependant, si nous avons deux ordinateurs, un qui a la capacité de faire tourner le jeu à 30 images par seconde et l'autre à 60 images par seconde; au bout de 2 secondes, le deuxième ordinateur aura fait avancer l'image deux fois plus loin que le premier. Pour résoudre ce problème, on peut utiliser la variable `delta` de la manière suivante :

```
batch.draw(image, positionX, 100);  
positionX += delta * 100;
```

De cette façon, l'image se déplacera toujours à la même vitesse, qu'importe la configuration de l'ordinateur.

3.3 Les Sprites

Pour déplacer une image, il est préférable d'utiliser des instances de *Sprite* comme précisé en introduction. Ainsi, l'image aura directement des fonctions de déplacement; le code précédant devient alors :



```

9  import com.badlogic.gdx.graphics.Texture;
10 import com.badlogic.gdx.graphics.g2d.Sprite;
11
12
13 public class Ecran implements Screen{
14
15     private BacASable game;
16     private Sprite image;
17
18     public Ecran(BacASable game) {
19         this.game = game;
20         image = new Sprite( new Texture("badlogic.jpg"));
21     }
22
23     @Override
24     public void render(float delta) {
25         Gdx.gl.glClearColor(1, 0, 0, 1);
26         Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
27
28         game.batch.begin();
29         image.draw(game.batch);
30         game.batch.end();
31     }
32

```

Figure 6 : Utilisation de la classe *Sprite*

3.4 Interaction avec l'utilisateur

Un jeu vidéo se caractérise avec son interaction avec l'utilisateur. Comment faire pour que notre image se déplace si et seulement le joueur appuie sur la touche A de son clavier? LibGDX fournit des outils permettant d'écouter les entrées claviers et souris que l'on peut utiliser de la manière suivante :



```

11
12
13 public class Ecran implements Screen{
14
15     private BacASable game;
16     private Sprite image;
17     private int speed = 30;
18
19     public Ecran(BacASable game) {
20         this.game = game;
21         image = new Sprite( new Texture("badlogic.jpg"));
22     }
23
24     @Override
25     public void render(float delta) {
26         Gdx.gl.glClearColor(1, 0, 0, 1);
27         Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
28
29         game.batch.begin();
30         image.draw(game.batch);
31         if (Gdx.input.isKeyPressed(Keys.A)) {
32             image.translateX(speed * delta);
33         }
34         game.batch.end();
35     }
36

```

Figure 6 : Gestion des inputs

Afin de simplifier la gestion des inputs lorsqu'il y a beaucoup de d'interactions différentes, il est préférable d'utiliser la classe *Controller* dont l'utilisation est détaillée sur le wiki officielle <https://github.com/libgdx/libgdx/wiki/Controllers>.

3.5 Pour aller plus loin

3.5.1 Fonctionnalités et extensions utiles

Afin de réaliser certaines tâches plus facilement, d'autres fonctions sont implémentées dans LibGDX, de nombreuses extensions existent également. On peut par exemple citer :

- la gestion des cartes <https://github.com/libgdx/libgdx/wiki/Tile-maps>,
- la gestion de la physique avec Box2D <https://github.com/libgdx/libgdx/wiki/Box2d>,
- la création d'interface textuelles avec Scene2D <https://github.com/libgdx/libgdx/wiki/Scene2d>.

3.5.2 Tutoriels

Afin de comprendre comme tous ces éléments fonctionnent, on peut trouver de très bons tutoriels sur Internet :

- le wiki officiel <https://github.com/libgdx/libgdx>,
- playlist de Brent Aureli <https://www.youtube.com/watch?v=a8MPxzkwBwo&list=PLZm85UZQLd2SXQzsF-a0-pPF6IWDDdrXt>.