Smokiev

Zeste de savoir

Un jeu de casse-briques en Lua avec Love2D

20 mars 2019

Table des matières

1. Introduction

Vous voulez apprendre à créer des jeux et vous avez entendu parler de Love2D ? Cela tombe bien, car à travers ce tutoriel, nous allons découvrir ce moteur de jeux 2D, multi-plateforme et totalement gratuit, tout en réalisant une implémentation du célèbre casse-briques ! Vous verrez ainsi qu'il est plutôt simple à prendre en main et puissant.

Comme son nom l'indique, le jeu de casse-briques consiste à détruire des briques. Pour cela, le joueur est muni d'une raquette lui permettant de frapper une balle. Lorsque celui-ci rate la balle, il perd une vie. Lorsqu'il n'a plus de vies, il perd la partie. Lorsque la balle entre en collision avec une brique, celle-ci est détruite. Le joueur gagne quand toutes les briques sont détruites.

Notez que ce tutoriel n'est qu'une introduction à ce moteur et ne le présente donc pas en profondeur. Ensuite, celui-ci s'utilisant avec le langage Lua, vous aurez sans doute besoin d'avoir des bases en celui-ci pour suivre sans problème.

Pour installer Love2D si ce n'est pas déjà fait, téléchargez le nécessaire sur cette page . La version actuelle, celle que j'utiliserai, est la 0.10.1. Par ailleurs, je me servirai de l'Environnement de Développement Intégré (EDI) ZeroBrane Studio pour développer. Si vous choisissez de l'utiliser, pensez à changer l'interpréteur pour l'exécution de votre projet (**Project > Lua Interpreter > LÖVE**).

Enfin, tout au long de votre lecture, n'hésitez pas à tester et à vous reporter à la documentation .

Prérequis
Bases en programmation
Connaissances en Lua (si besoin, référez-vous à ce tutoriel ou encore à celui-ci)
Objectifs

Faire découvrir Love2D
Réaliser un jeu de casse-briques

2. Première fenêtre

Commençons par créer notre première fenêtre.

2.0.1. Callbacks et squelette de base

Tout d'abord, Love2D contient ce que l'on appelle des fonctions de rappel (*callback*). Celles-ci sont appelées automatiquement et dans un ordre bien précis lors de l'exécution du jeu, si elles sont définies.

Ainsi, il y a par exemple, la fonction `love.load` qui est appelée au début et qui a pour but de charger les ressources (images, sons, etc.) ou encore d'initialiser des variables. De même, il existe une fonction `love.draw` destinée à dessiner à l'écran.

Aussi, certaines d'entre elles peuvent prendre des paramètres, comme la fonction `love.update` par exemple. Celle-ci prend le temps écoulé depuis le dernier appel à elle : le *deltatime*. De cette manière, il est possible de mettre à jour un jeu (la position d'un personnage par exemple) indépendamment de la puissance de l'ordinateur (l'affichage ne dépend pas de l'exécution d'une boucle).

Avec celles-ci ainsi que la fonction `love.keypressed` qui permet de gérer l'appui sur une touche du clavier, nous arrivons à ce squelette de base :

Fichier main.lua

```lua
-- pour écrire dans la console au fur et à mesure, facilitant ainsi le débogage
io.stdout:setvbuf('no')

function love.load()
  -- Fonction pour initialiser le jeu (appelée au début de celui-ci)
end

function love.update(dt)
  -- Fonction pour mettre à jour (appelée à chaque frame)
end

function love.draw()
  -- Fonction pour dessiner (appelée à chaque frame)
end

function love.keypressed(key)
```

```
17    -- Fonction pour gérer l'appui sur les touches (appelée pour
          chaque touche pressée)
18  end
```

En exécutant ce code, vous devriez voir votre première fenêtre s'ouvrir ! Certes, il n'y a pas grand chose d'intéressant pour le moment, mais patience !

Si vous cherchez à en savoir un peu plus sur ces fonctions et voir dans quel ordre elles s'organisent, je vous renvoie respectivement à ce lien et à celui-ci .

2.0.2. Configuration

Avant d'aller plus loin, nous allons d'abord configurer un peu notre fenêtre. Nous allons donc lui ajouter un titre ainsi qu'une icône et choisir ses dimensions.

Pour l'icône, enregistrez l'image ci-dessous dans un répertoire « **images** » sous le nom « **icon.png** », dans le répertoire de votre projet.

FIGURE 2. – L'icône à télécharger.

2.0.2.1. Avec des fonctions spécifiques Pour paramétrer tout ça, nous pouvons utiliser des fonctions du module `love.window` que vous pouvez explorer ici . Faisons donc appel à celles-ci à l'intérieur de `love.load`, donc au début de l'exécution de notre jeu :

Fichier main.lua

```
1  function love.load()
2
3    love.window.setTitle("Casse-briques") -- Change le titre de la
         fenêtre
4    local imgIcon = love.graphics.newImage("images/icon.png") --
         Chargement de l'image
5    love.window.setIcon(imgIcon:getData()) -- Change l'icone de la
         fenêtre
6    love.window.setMode(480, 640) -- Change les dimensions de la
         fenêtre
7
8  end
```

Comme vous pouvez le constater, nous changeons d'abord le titre avec `love.window.setTitle`. Ensuite, nous chargeons l'image de notre icône que nous passons au bon format à `love.window.setIcon`. Enfin, nous terminons en changeant les dimensions de la fenêtre en passant la largeur puis la hauteur de celle-ci à `love.window.setMode`.

2.0.2.2. Avec un fichier conf.lua

Alternativement, nous pouvons aussi passer par un fichier **conf.lua** qui contiendra une fonction `love.conf`. Comme vous l'avez compris, celle-ci est effectivement une fonction de *callback*. Elle est appelée au lancement du jeu avant même le `love.load` et sert à configurer celui-ci à l'aide d'une variable reçue en paramètre.

Ainsi, en l'utilisant, cela donne :

Fichier conf.lua

```lua
function love.conf(t)

  t.window.title = "Casse-briques" -- Change le titre de la fenêtre
  t.window.icon = "images/icon.png" -- Change l'icone de la fenêtre
  t.window.width = 480 -- Change la largeur de la fenêtre
  t.window.height = 640 -- Change la hauteur de la fenêtre

end
```

Pour en savoir plus sur cette fonction, vous pouvez jeter un coup d'œil ici .

Voilà, à vous de choisir entre ces deux approches. Pour ma part, je préfère la seconde, parce qu'elle est effectuée en première et parce que ça libère de la place dans le `love.load`.

2.0.3. Constantes

Pour finir, afin d'avoir un code plus modulable, ajoutons un fichier **constants.lua** qui contiendra les constantes de notre jeu. Voici ce que ça donne :

Fichier constants.lua

```lua
TITLE = "Casse-briques" -- Titre
PATH_ICON = "images/icon.png" -- Chemin image icône
WIN_WIDTH = 480 -- Largeur fenêtre
WIN_HEIGHT = 640 -- Hauteur fenêtre
```

Il ne vous reste plus qu'à remplacer les valeurs dans `love.load` ou `love.conf` par ces variables après les avoir importées :

```lua
require('constants')
```

De la sorte, si nous souhaitons modifier une valeur, nous n'aurons qu'à jeter un coup d'œil au fichier des constantes et non à chercher dans tout le code. Cela est très utile quand la valeur en question apparaît à de nombreuses reprises par exemple.

Au terme de cette section, vous devriez avoir une fenêtre qui ressemble à celle présentée ci-dessous. Pour ma part, je n'ai pas l'icône d'affichée (sans doute parce que je suis sous *Ubuntu*).

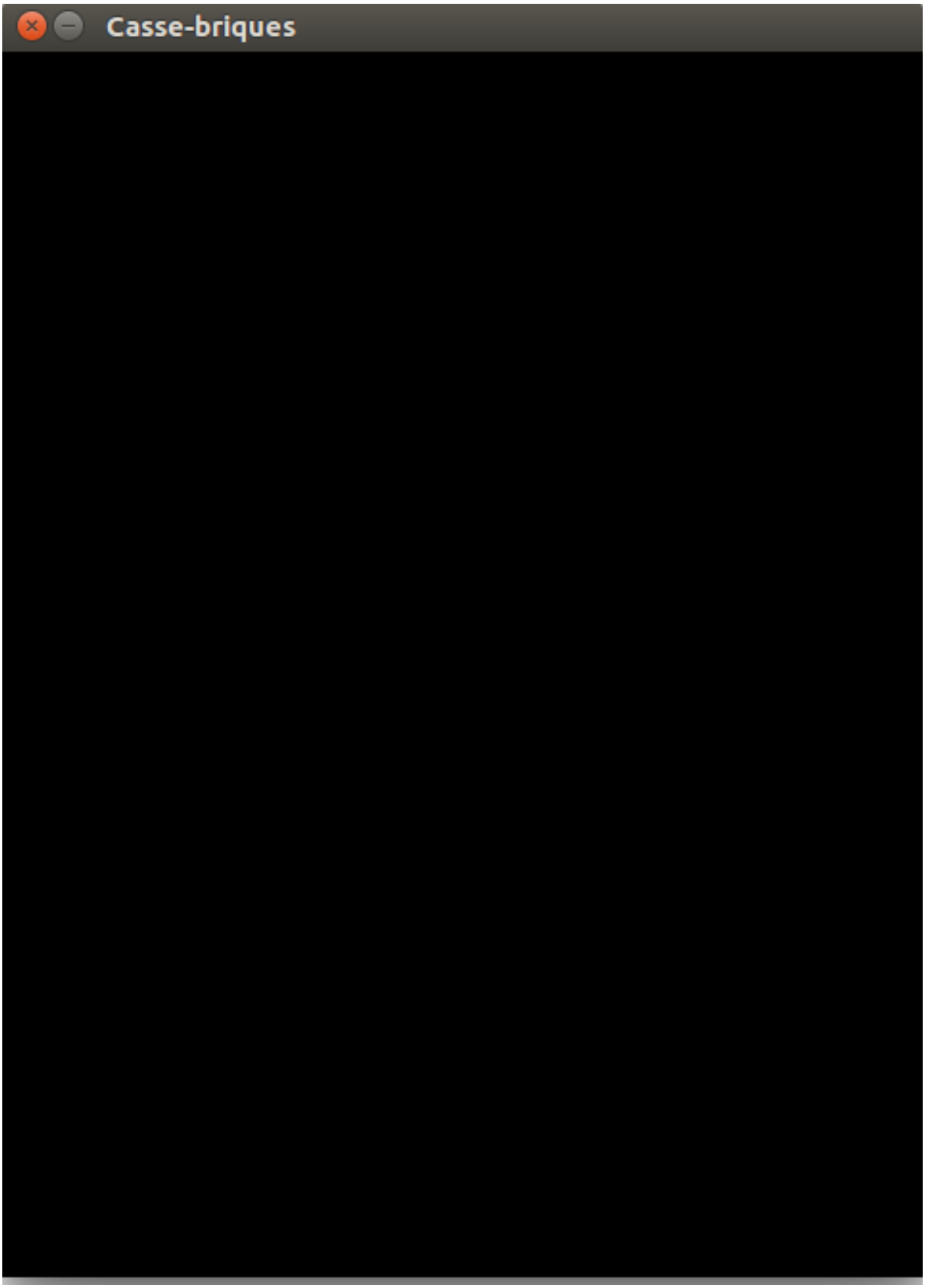

FIGURE 2. – Notre première fenêtre.

Nous allons maintenant ajouter les premiers éléments au jeu.

3. La raquette et les briques

Oui, il est temps de passer aux choses sérieuses ! Dans cette section, nous allons mettre en place la raquette et les briques. Nous allons donc les afficher et faire bouger la première.

Petit rappel si vous n'êtes pas familier avec les bibliothèques de jeux 2D, sachez que très souvent l'origine du repère des coordonnées se trouve dans le coin haut gauche. De plus, l'axe des abscisses est orienté vers la droite tandis que l'axe des ordonnées est orienté vers le bas. Ainsi, les coordonnées du coin haut gauche sont *(0, 0)* tandis que celles du coin bas droit sont *(largeur, hauteur)*. Par ailleurs, les éléments sont situés avec la position de leur coin haut gauche. Par exemple, si notre raquette a une position en y élevée, elle sera donc proche du bas de la fenêtre. De même, si sa position en x vaut 0, alors elle touchera le bord gauche de la fenêtre.

Voici un exemple d'illustration d'un rectangle en *(posX, posY)* pour bien comprendre :

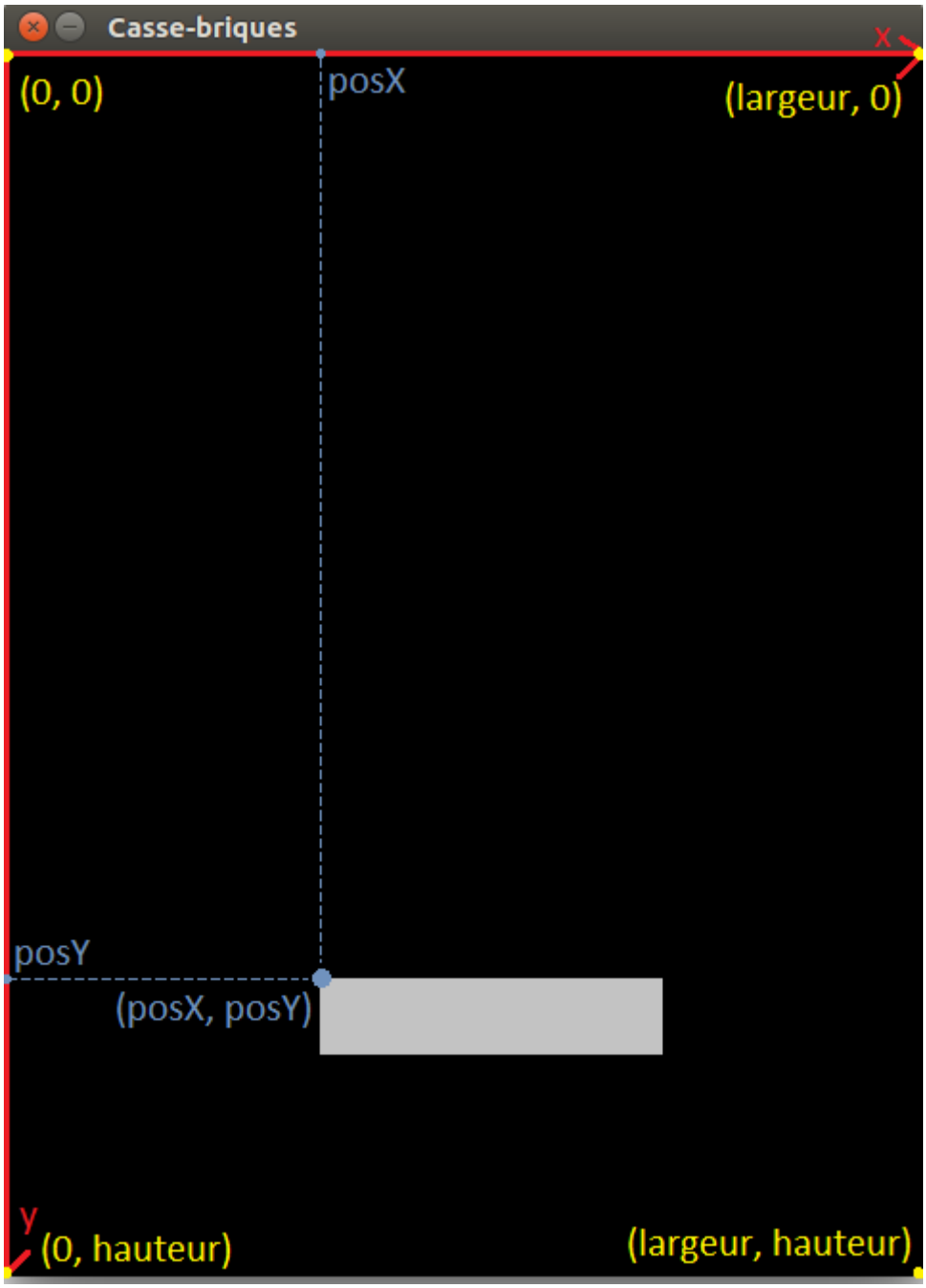

FIGURE 3. – Repère de coordonnées.

3.0.1. La raquette

Démarrons avec la raquette. Celle-ci sera placée vers le bas de la fenêtre. De plus, elle se déplacera latéralement en fonction des touches pressées (vers la gauche avec `Q` ou `<-` et vers la droite avec `D` ou `->`).

3.0.1.1. Déclaration et initialisation Pour mettre à jour le jeu, il nous faut conserver certaines caractéristiques de la raquette. En effet, nous voulons pouvoir connaître ses dimensions, savoir où elle se trouve à un moment précis ou encore connaître sa vitesse. De même, nous voulons pouvoir modifier aisément ces valeurs si besoin, pour modifier la position de la raquette par

exemple. Pour cela, nous allons utiliser une table et l'initialiser avec une fonction de notre cru appelée dans `love.load` :

```lua
local racket -- Déclaration variable pour la raquette

function initializeRacket()

  racket = {} -- Initialisation variable pour la raquette

  -- Initialisation de paires (clef, valeur) de la table racket
  racket.speedX = 215 -- Vitesse horizontale
  racket.width = WIN_WIDTH / 4 -- Largeur
  racket.height = WIN_HEIGHT / 37 -- Hauteur
  racket.x = (WIN_WIDTH-racket.width) / 2 -- Position en abscisse
  racket.y = WIN_HEIGHT - 64 -- Position en ordonnée

end

function love.load()
  initializeRacket()
end
```

Comme vous pouvez le voir, nous définissons la taille et la position de notre raquette en fonction des dimensions de la fenêtre. De cette manière, si nous changeons ces dernières dans les constantes, le jeu sera affiché de la même façon. Nous procéderons donc ainsi tout au long de ce tutoriel.

3.0.1.2. Affichage La raquette étant créée, nous pouvons la dessiner dans `love.draw`. Ainsi, nous choisissons d'abord la couleur blanche puis nous dessinons le rectangle correspondant, respectivement avec les fonctions `love.graphics.setColor` et `love.graphics.rectangle`. Pour la couleur, nous la passons sous la forme RGB *(Red, Green, Blue)*. Pour le rectangle, nous indiquons en premier paramètre que nous le voulons rempli et ensuite nous passons les coordonnées et les dimensions.

```lua
function love.draw()
  love.graphics.setColor(255, 255, 255) -- Couleur blanche
  love.graphics.rectangle('fill', racket.x, racket.y, racket.width,
      racket.height) -- Rectangle
end
```

Comme vous pourrez le constatez si vous regardez la documentation, le module love.graphics ↗ contient de nombreuses fonctions pour dessiner à l'écran.

Eh voilà ! Si vous exécutez le code, vous verrez bien une raquette apparaître à l'écran. Désormais, il ne nous reste plus qu'à la faire bouger.

FIGURE 3. – La raquette.

3.0.1.3. Déplacement Pour faire mouvoir notre raquette, nous allons devoir récupérer des événements en provenance du clavier. Là, si vous avez bien suivi, vous penserez sans doute que nous allons utiliser la fonction `love.keypressed`. Eh bien, que nenni ! En effet, si nous utilisions cette dernière, nous devrions appuyer sans cesse sur les touches pour nous déplacer (par défaut, il n'y pas de fluidité en restant appuyé) et il nous manquerait le fameux *deltatime*. Pour résoudre cela, nous allons donc compléter notre `love.update`, car vu que cette fonction est appelée de nombreuses fois par seconde, l'appui sur une touche donnera une impression de fluidité dans le déplacement.

Ainsi, nous testons si les touches que nous voulons sont actuellement pressées avec `love.keyboard.isDown` du module love.keyboard , et nous modifions la position de la raquette de sorte qu'elle ne sorte pas de l'écran :

```
1  function love.update(dt)
2
3    -- Mouvement vers la gauche
4    if love.keyboard.isDown('left', 'q') and racket.x > 0 then
5        racket.x = racket.x - (racket.speedX*dt)
6    -- Mouvement vers la droite
7    elseif love.keyboard.isDown('right', 'd') and racket.x +
         racket.width < WIN_WIDTH then
8        racket.x = racket.x + (racket.speedX*dt)
9    end
10
11 end
```

Ça y est, notre raquette est opérationnelle.

Les briques

Au tour des briques maintenant. Nous les placerons en haut de notre fenêtre.

3.0.1.4. Déclaration et initialisation

Comme pour la raquette, les briques ont certaines caractéristiques que nous devons stocker et comme pour la raquette nous allons utiliser une table ainsi qu'une fonction pour initialiser celle-ci.

Ajoutons donc la ligne suivante en dessous de la déclaration de *racket* :

```
local bricks -- Déclaration variable pour les briques
```

Créons aussi deux constantes dans le fichier **constants.lua** pour qu'il soit facile de faire varier le nombre de briques :

```
BRICKS_PER_LINE = 7 -- Nombre de briques par ligne
BRICKS_PER_COLUMN = 6 -- Nombre de briques par colonne
```

Puis initialisons les briques :

```
function createBrick(line, column)

  -- Fonction pour créer une brique et l'initialiser en fonction de
      sa position dans le mur
  local brick = {}
  brick.isNotBroken = true -- Brique pas encore cassée
  brick.width = WIN_WIDTH / BRICKS_PER_LINE - 5 -- Largeur
  brick.height = WIN_HEIGHT / 35 -- Hauteur
  brick.x = 2.5 + (column-1) * (5+brick.width) -- Position en
      abscisse
  brick.y = line * (WIN_HEIGHT/35+2.5) -- Position en ordonnée
  return brick

end

function initializeBricks()

    bricks = {} -- Initialisation variable pour les briques
    for line=1, BRICKS_PER_COLUMN do
      table.insert(bricks, {}) -- Ajout d'une ligne
      for column=1, BRICKS_PER_LINE do
        local brick = createBrick(line, column)
        table.insert(bricks[line], brick) -- Ajout d'une brique par
            colonne de la ligne
      end
    end

end
```

Comme vous pouvez le voir, nous avons une fonction permettant de créer une brique qui sera appelée lors de l'initialisation pour chaque emplacement du mur de briques.

N'oubliez pas de faire appel à `initializeBricks` dans `love.load` sans quoi la table des briques ne sera pas créée.

3.0.1.5. Affichage Enfin, il ne nous reste plus qu'à dessiner nos briques dans `love.draw` avec des rectangles comme nous l'avons fait pour la raquette. Pour cela, nous parcourons notre tableau à deux dimensions et nous faisons appel à `love.graphics.rectangle` pour chaque brique non cassée.

```
for line=1, #bricks do -- Ligne
  for column=1, #bricks[line] do -- Colonne
    local brick = bricks[line][column]
    if brick.isNotBroken then -- Si la brique n'est pas cassée
      love.graphics.rectangle('fill', brick.x, brick.y,
          brick.width, brick.height) -- Rectangle
    end
  end
end
```

Comme convenu, nos briques s'affichent à l'exécution :

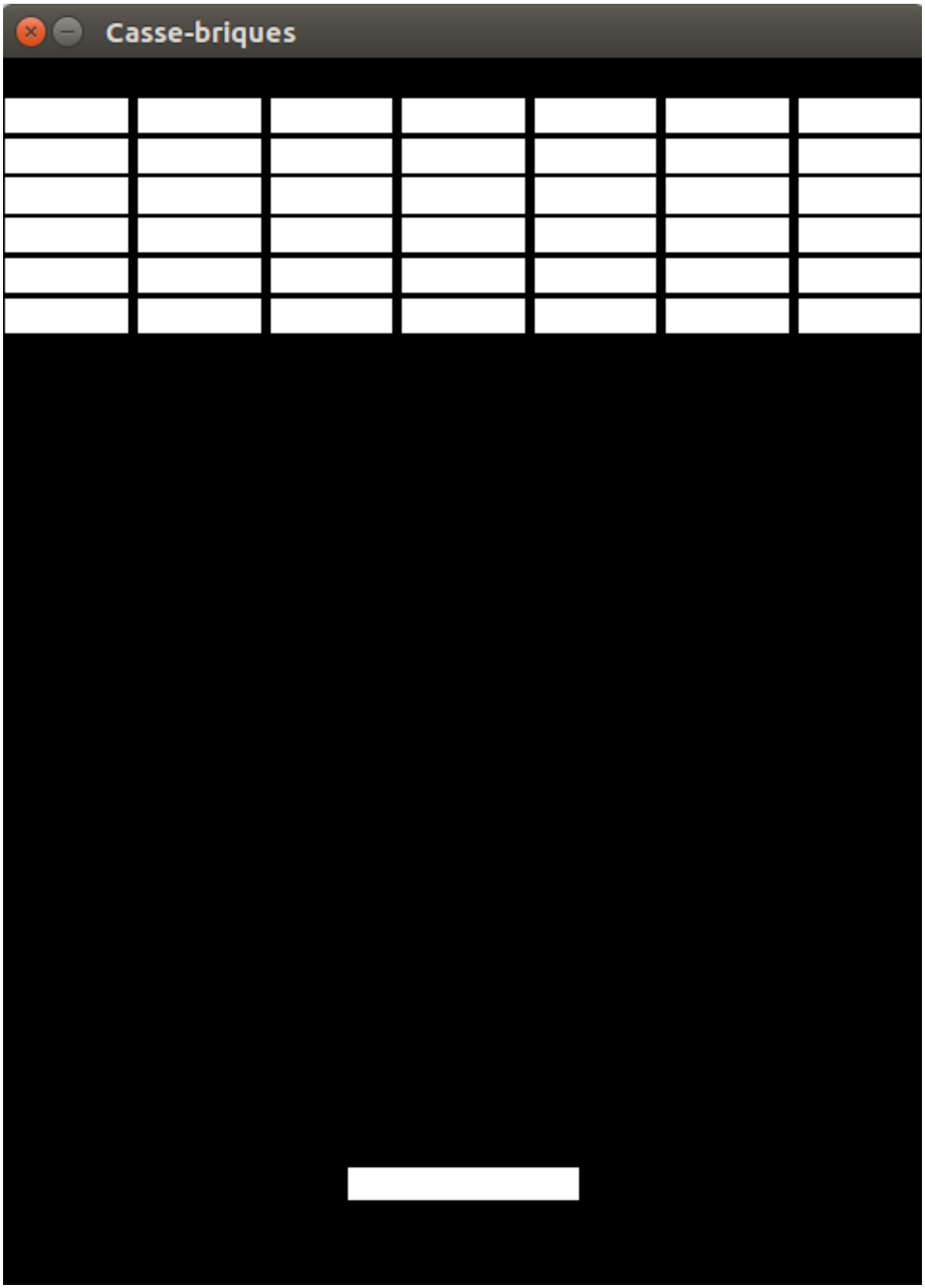

FIGURE 3. – Les briques sont là !

Le jeu commence à prendre forme, n'est-ce pas ? Toutefois, ne nous arrêtons pas en si bon chemin !

4. Les vies et la balle

C'est bien sympa, mais vous admettrez qu'on s'ennuie un peu pour l'instant, non ? Qu'à cela ne tienne, nous allons rajouter un système de vie et une balle !

4.0.1. Les vies

Attaquons avec les vies. Pour notre jeu, le joueur en aura trois au départ et nous afficherons celles-ci en bas à gauche de la fenêtre.

Voici l'image que nous utiliserons. Enregistrez la sous le nom « **life.png** », dans le même répertoire que l'icône.

FIGURE 4. – Image d'une vie.

4.0.1.1. Déclaration et initialisation Une fois n'est pas coutume, nous allons utiliser une table pour gérer les vies :

```
1 local lives -- Déclaration variable pour les vies
```

Et nous allons initialiser cette table avec une fonction appelée dans `love.load` :

```
1 function initializeLives()
2
3   lives = {} -- Initialisation variable pour les vies
4   lives.count = NB_LIVES -- Nombre de vie
5   lives.img = love.graphics.newImage(PATH_LIFE) -- Image vie
6   lives.width, lives.height = lives.img:getDimensions() --
        Dimensions de l'image
7
8 end
```

Comme vous pouvez le voir, nous chargeons l'image en faisant appel à `love.graphics.newImage` avec le chemin de celle-ci en paramètre (si vous avez fait attention, nous avions déjà utilisée cette fonction pour l'icône). De plus, nous nous servons de notre image en faisant appel à `getDimensions` afin de récupérer la largeur et la hauteur de celle-ci.

Par ailleurs, comme vous le devinez, il faut ajouter quelques constantes au fichier **constants.lua** pour que ça fonctionne :

```lua
NB_LIVES = 3 -- Nombre de vies initiales
PATH_LIFE = "images/life.png" -- Chemin image vie
```

4.0.1.2. Affichage Désormais, il ne reste plus qu'à dessiner chaque vie dans `love.draw`. Dans ce but, nous allons utiliser la fonction `love.graphics.draw` en lui passant l'image à dessiner ainsi que la position en abscisse et la position en ordonnée :

```lua
for i=0, lives.count-1 do -- Pour chaque vie
    local posX = 5 + i * 1.20 * lives.width -- Calcul de la
        position en abscisse
    love.graphics.draw(lives.img, posX, WIN_HEIGHT-lives.height) --
        Affichage de l'image
end
```

Si tout se passe bien, vous verrez les vies apparaître comme désiré :

FIGURE 4. – Les vies.

La balle

Il ne nous reste plus que la balle pour que le jeu soit jouable. Celle-ci sera carrée, apparaîtra juste au dessus du centre de la raquette et aura une direction initiale aléatoire.

4.0.1.3. Déclaration et initialisation Vous commencez à être rodé avec le système : nous allons créer une table spécifique et nous allons initialiser celle-ci au chargement du jeu :

```lua
local ball -- Déclaration variable pour la balle
```

Mais contrairement à d'habitude, nous allons passer deux paramètres à la fonction d'initialisation afin que la balle soit proportionnelle à la raquette et soit placée au dessus :

```lua
function initializeBall(racketHeight, racketY)

  ball = {} -- Initialisation variable pour la balle
  ball.width, ball.height = racketHeight * 0.75, racketHeight * 0.75  -- Taille
  ball.speedY = -DEFAULT_SPEED_BY -- Vitesse verticale
  ball.speedX = math.random(-DEFAULT_SPEED_BX, DEFAULT_SPEED_BX) -- Vitesse horizontale
  ball.x = WIN_WIDTH / 2 - ball.width / 2 -- Position en abscisse
  ball.y = racketY - 2 * ball.height - ball.height / 2 -- Position en ordonnée

end
```

De cette manière, si nous changeons la hauteur ou la position en ordonnée de la raquette, la balle sera toujours affichée correctement. D'ailleurs, n'oubliez pas de l'initialiser dans `love.load` :

```lua
initializeBall(racket.height, racket.y)
```

Ensuite, vous avez dû vous rendre compte qu'il fallait ajouter quelques constantes :

```lua
DEFAULT_SPEED_BX = 130 -- Vitesse horizontale
DEFAULT_SPEED_BY = 335 -- Vitesse verticale
```

? Et puis, qu'est-ce que ce `math.random` ?

Eh bien, celui-ci nous sert à choisir aléatoirement une vitesse en abscisse pour la balle dans intervalle [-DEFAULT_SPEED_BX, DEFAULT_SPEED_BX]. Comme Lua charge automatiquement les bibliothèques standards dans l'environnement global, nous n'avons même pas besoin d'importer `math` avant de l'utiliser.

Enfin, si vous connaissez déjà un peu les générateurs de nombres pseudo-aléatoires, vous savez qu'il nous faut initialiser la graine aléatoire avec une valeur qui ne sera jamais identique, sans quoi la suite de résultats sera toujours la même (par exemple, la première balle partira à gauche, la seconde à droite, etc. à chaque partie) :

```lua
math.randomseed(love.timer.getTime()) -- Initialisation de la graine avec un temps en ms
```

Nous placerons la ligne précédente au début de `love.load`.

4.0.1.4. Affichage L'affichage de la balle est maintenant très simple puisqu'il suffit de dessiner le rectangle adéquate dans `love.draw`, comme nous avons déjà pu le faire :

```
love.graphics.rectangle('fill', ball.x, ball.y, ball.width,
    ball.height) -- Rectangle
```

FIGURE 4. – Apparition de la balle.

4.0.1.5. Déplacement et collisions Voici sans doute la partie la plus intéressante puisqu'il va y avoir enfin de l'action impliquant des conséquences dans notre jeu. En effet, jusqu'à présent, seule notre raquette bougeait, et encore, il fallait appuyer sur des touches pour cela. Or, la balle va se déplacer toute seule. De plus, elle va entrer en collision avec son environnement.

Afin de tester la collision entre deux rectangles (sans rotation dans le plan), nous utiliserons la fonction ci-dessous tirée d'ici , où il est expliqué qu'elle vérifie s'il n'y a pas du vide autour des 4 côtés du rectangle. S'il n'y a pas que du vide, c'est donc qu'il y a une collision. Comme nous pouvons le voir, les deux premières conditions permettent de tester la collision en abscisse (par la gauche ou par la droite) tandis que les deux secondes s'occupent de la collision en ordonnée (par le haut ou par le bas). Pour vous approprier le fonctionnement du code, vous pouvez l'essayer ici .

```
function collideRect(rect1, rect2)
  if rect1.x < rect2.x + rect2.width and
     rect1.x + rect1.width > rect2.x and
     rect1.y < rect2.y + rect2.height and
     rect1.height + rect1.y > rect2.y then
       return true
  end
  return false
end
```

Vous pouvez d'ores et déjà ajouter cette fonction au fichier **constants.lua**.

Déplacement Pour déplacer notre balle, il nous suffit de modifier sa position en fonction du temps écoulé, dans `love.update` :

```
ball.x = ball.x + ball.speedX * dt -- Mise à jour position en abscisse de la balle
ball.y = ball.y + ball.speedY * dt -- Mise à jour position en ordonnée de la balle
```

En exécutant vous verrez, ô miracle, la balle se déplacer, et ... sortir de la fenêtre.

Collisions avec la fenêtre

En fait, à bien y réfléchir, c'est tout à fait normal puisque nous diminuons à chaque rafraîchissement la position en ordonnée de la balle. Du coup, elle finit par passer la bordure du haut. De la même façon, si le déplacement en x est non nul, la balle finira par dépasser par la gauche ou par la droite de la fenêtre.

Vous l'aurez compris, il faut que notre balle puisse rebondir sur les bordures afin de ne pas quitter l'écran. À une exception près : si la balle sort par le bas alors c'est que la raquette ne l'a pas renvoyée, donc nous devons enlever une vie au joueur et réinitialiser la balle.

Il faut donc rajouter ceci à `love.update` :

```
if ball.x + ball.width >= WIN_WIDTH then  -- Bordure droite
  ball.speedX = -ball.speedX
elseif ball.x <= 0 then -- Bordure gauche
  ball.speedX = -ball.speedX
end

if ball.y <= 0 then  -- Bordure haut
  ball.speedY = -ball.speedY
elseif ball.y + ball.height >= WIN_HEIGHT then -- Bordure bas
  lives.count = lives.count - 1
  resetBall(racket.y)
end
```

Et voici la fonction pour réinitialiser la balle :

```
function resetBall(racketY)

  ball.speedY = -DEFAULT_SPEED_BY -- Vitesse verticale
  ball.speedX = math.random(-DEFAULT_SPEED_BX, DEFAULT_SPEED_BX) -- Vitesse horizontale
  ball.x = WIN_WIDTH / 2 - ball.width / 2 -- Position en abscisse
  ball.y = racketY - 2 * ball.height - ball.height / 2 -- Position en ordonnée

end
```

Vous remarquerez qu'elle reprend en partie `initializeBall`, donc nous pourrions remplacer les lignes redondantes dans cette dernière par un appel à cette fonction, afin d'éviter la duplication de code.

Collisions avec la raquette Pour tester la collision entre la raquette et la balle, il nous suffit de tester la collision dans `love.update` en faisant appel à `collideRect` définie précédemment :

```
if collideRect(ball, racket) then
  collisionBallWithRacket() -- Collision entre la balle et la raquette
end
```

Et de changer la direction de la balle en fonction de la collision :

```
function collisionBallWithRacket()

    -- Collision par la gauche (coin haut inclus)
    if ball.x < racket.x + 1/8 * racket.width and ball.speedX >= 0 then
      if ball.speedX <= DEFAULT_SPEED_BX/2 then -- Si vitesse trop faible
        ball.speedX = -math.random(0.75*DEFAULT_SPEED_BX, DEFAULT_SPEED_BX) -- Nouvelle vitesse
      else
        ball.speedX = -ball.speedX
      end
    -- Collision par la droite (coin haut inclus)
    elseif ball.x > racket.x + 7/8 * racket.width and ball.speedX <= 0 then
      if ball.speedX >= -DEFAULT_SPEED_BX/2 then  -- Si vitesse trop faible
        ball.speedX = math.random(0.75*DEFAULT_SPEED_BX, DEFAULT_SPEED_BX) -- Nouvelle vitesse
      else
        ball.speedX = -ball.speedX
      end
    end
    -- Collision par le haut
    if ball.y < racket.y and ball.speedY > 0 then
      ball.speedY = -ball.speedY
  end

end
```

Vous pouvez remarquez que lors de la collision avec les côtés et les coins du haut de la raquette, nous en profitons pour donner une nouvelle vitesse de déplacement en abscisse à la balle si

celle-ci est trop faible. Cela peut donne une impression d'effet et permet surtout d'éviter que la balle fasse du surplace en x.

Collisions avec les briques Pour la collision avec les briques, nous allons rajouter une variable locale en dessous des autres afin de voir facilement combien de briques il reste, ce qui nous servira par la suite pour tester la fin du jeu.

```
local nbBricks = BRICKS_PER_COLUMN * BRICKS_PER_LINE -- Nombre de
    briques
```

Comme pour la raquette, nous allons devoir tester la collision dans `love.update` :

```
for line=#bricks, 1, -1 do
  for column=#bricks[line], 1, -1 do
    if bricks[line][column].isNotBroken and collideRect(ball,
        bricks[line][column]) then
      collisionBallWithBrick(ball, bricks[line][column]) --
          Collision entre la balle et une brique
    end
  end
end
```

Et modifier le jeu en fonction (modification de la direction de la balle et suppression de la brique) :

```
function collisionBallWithBrick(ball, brick)

  -- Collision côté gauche brique
  if ball.x < brick.x and ball.speedX > 0 then
      ball.speedX = -ball.speedX
  -- Collision côté droit brique
  elseif ball.x > brick.x + brick.width and ball.speedX < 0 then
      ball.speedX = -ball.speedX
  end
  -- collision haut brique
  if ball.y < brick.y and ball.speedY > 0 then
    ball.speedY = -ball.speedY
  -- Collision bas brique
  elseif ball.y > brick.y and ball.speedY < 0 then
    ball.speedY = -ball.speedY
  end

  brick.isNotBroken = false -- Brique maintenant cassée
```

```
19    nbBricks = nbBricks - 1 -- Ne pas oublier de décrémenter le
          nombre de briques
20
21  end
```

Si tout se passe bien, vous devriez pouvoir vous amusez un peu :

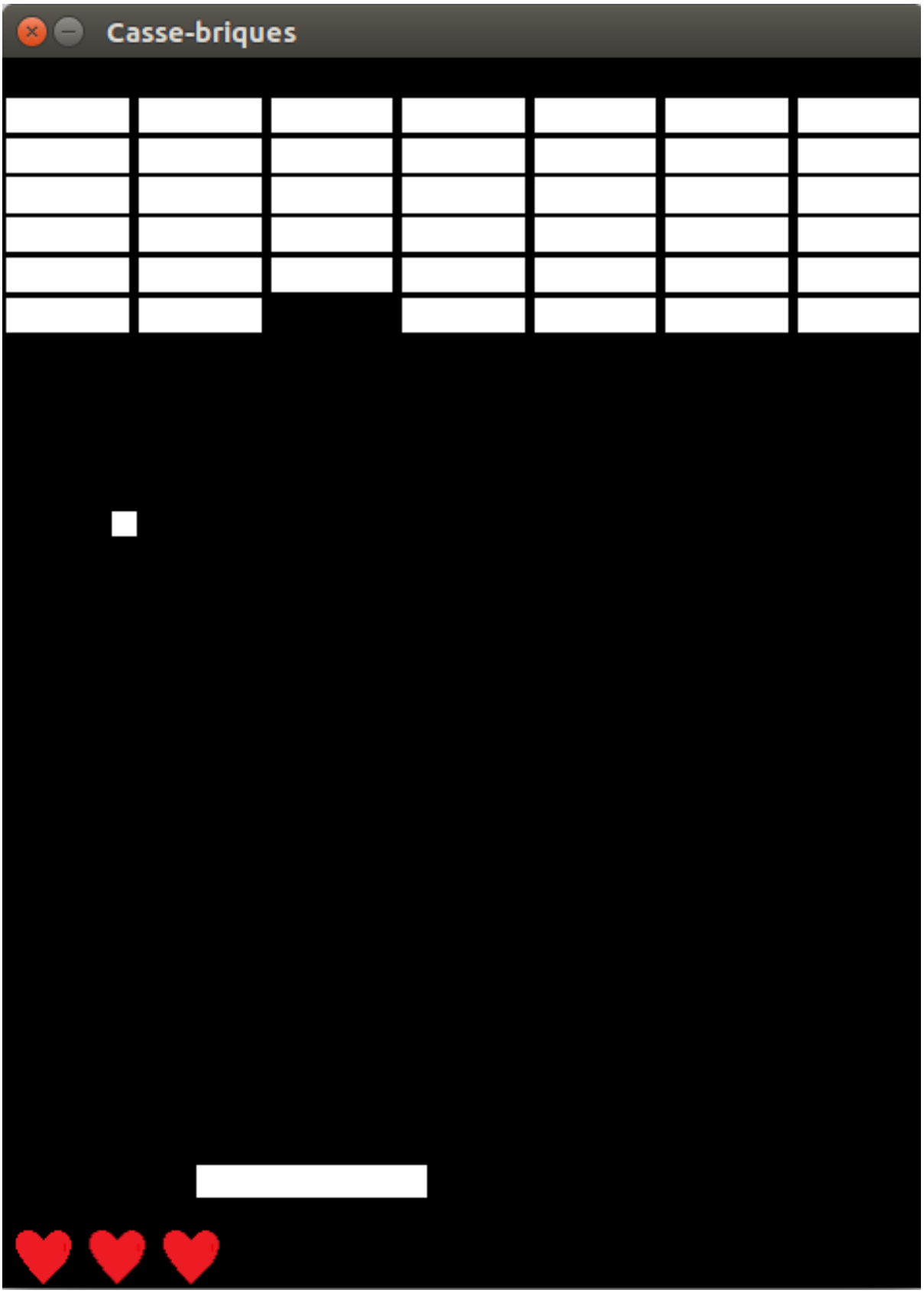

FIGURE 4. – Prêt à tout casser ?

Désormais, nous avons quasiment tout ce qu'il faut pour un jeu fonctionnel ! Il nous reste à tester la fin du jeu, mais avant cela nous allons rajouter quelques sons.

5. Les sons

Le module love.audio va nous permettre de mettre en place les sons. Avant cela, je vous laisse télécharger ce son que nous jouerons lorsque la balle touchera une brique ainsi que celui que nous jouerons lorsque la raquette frappera la balle. Placez les dans un répertoire « **sounds** » à l'intérieur de votre projet.

Pour information, ceux-ci ont été réalisés avec Bfxr , un outil de génération de sons bien pratique.

5.0.0.1. Charger les sons

La fonction `love.audio.newSource` permet de charger un son en lui spécifiant son chemin. Par ailleurs, en précisant `"static"` en second paramètre, nous pouvons indiquer que nous voulons charger le son directement en mémoire. Dans le cas contraire, le fichier sera chargé au fur et à mesure de la lecture. Pour un fichier plutôt lourd telle qu'une musique, il est conseillée d'opter pour cette seconde option, de sorte à ne pas encombrer la *RAM*.

Commençons par déclarer deux variables locales au début du code :

```
1 local soundBrick -- Déclaration variable son brique
2 local soundRacket -- Déclaration variable son raquette
```

Puis chargeons nos sons dans `love.load` :

```
1 soundBrick = love.audio.newSource(PATH_SOUND_BRICK, "static") --
      Chargement son brique
2 soundRacket = love.audio.newSource(PATH_SOUND_RACKET, "static") --
      Chargement son raquette
```

Bien sûr, il nous faut créer les constantes dans le fichier **constants.lua** pour que cela fonctionne :

```
1 PATH_SOUND_BRICK = "sounds/collision_brick.wav" -- Chemin son
      brique
2 PATH_SOUND_RACKET = "sounds/collision_racket.wav" -- Chemin son
      raquette
```

5.0.0.2. Jouer les sons

Pour jouer un son, il suffit de s'en servir en faisant appel à `play`. Ainsi, pour jouer le son de la collision entre la balle et la raquette dans `collisionBallWithRacket` et pour jouer le son de la collision entre la balle et une brique dans `collisionBallWithBrick`, nous mettrons respectivement au début de ces fonctions :

```lua
soundRacket:play() -- Joue le son raquette
```

```lua
soundBrick:play() -- Joue le son brique
```

Alors, c'est tout de suite plus vivant avec un peu de son, non ?

6. Le menu

À travers cette section, nous allons ajouter un menu et enfin achever le jeu !

Pour commencer, nous allons organiser ce dernier en trois pages :

— une page de début s'affichant avant de lancer la partie ;
— une page de partie ;
— une page de fin s'affichant une fois la partie terminée.

Rajoutons donc trois constantes dans **constants.lua** :

```lua
PAGE_BEGINNING = 1 -- Page de début
PAGE_ROUND = 2 -- Page de partie
PAGE_END = 3 -- Page de fin
```

Et créons une variable locale dans **main.lua** pour garder la page courante :

```lua
currentPage = PAGE_BEGINNING -- Page courante
```

Enfin, il nous faut réorganiser notre `love.draw` et notre `love.update` pour n'afficher et ne mettre à jour que la page courante (il ne faudrait pas que notre partie s'affiche ou se déroule alors que nous sommes sur la page d'accueil par exemple). Cela se fait bêtement de la sorte :

```lua
if currentPage == PAGE_BEGINNING then
  -- Traitement page début
elseif currentPage == PAGE_ROUND then
  -- Traitement page partie : placer le code déjà présent ici.
elseif currentPage == PAGE_END then
  -- Traitement page fin
end
```

Au passage, créons une fonction `drawRound` ainsi qu'une fonction `updateRound` qui contiendront respectivement le code pour afficher et mettre à jour la partie. Nous allons appeler celles-ci

respectivement dans `love.draw` et `love.update` quand la page courante vaut `PAGE_ROUND`. De cette manière, nous factorisons le code et, surtout, gagnons en visibilité.

N'oubliez pas de passer le *deltatime* en paramètre à `updateRound`, sans quoi la partie ne pourra pas se mettre à jour.

6.0.1. Première page

Notre première page affichera le titre et indiquera comment lancer la partie. Pour cela, nous allons créer une police d'écriture et écrire dans la fenêtre. Tout d'abord, déclarons une variable locale au début :

```
1 local font -- Déclaration variable pour la police d'écriture
```

Puis initialisons la dans `love.load` en appelant `love.graphics.newFont` avec une taille en paramètre :

```
1 font = love.graphics.newFont(32) -- Initialise font avec une police
      de taille 32
```

De plus, pour se servir d'une de nos polices, nous devons l'indiquer avec `love.graphics.setFont`. Vu que nous n'utiliserons que celle-ci, nous pouvons donc ajouter la ligne suivante à la suite :

```
1 love.graphics.setFont(font) -- Définit font comme la police
      utilisée
```

Maintenant il ne reste plus qu'à écrire dans `love.draw` en utilisant la fonction `love.graphics.printf`. Celle-ci prend en paramètre le texte à écrire, la position en abscisse puis la position en ordonnée de celui-ci ainsi que la largeur de la *boîte* dans laquelle sera écrit le texte. Si cette dernière est plus petite que la longueur nécessaire pour écrire le texte, alors il sera écrit sur plusieurs lignes. Par ailleurs, nous pouvons indiquer un alignement dans cette boîte en dernier paramètre. Ainsi, nous allons centrer nos textes en largeur de la sorte :

```
1 love.graphics.printf("Casse-briques", 0, 0.25*WIN_HEIGHT,
      WIN_WIDTH, "center") -- Écriture
2 love.graphics.printf("Appuyez sur 'R' pour commencer", 0,
      0.45*WIN_HEIGHT, WIN_WIDTH, "center") -- Écriture
```

Cela donne :

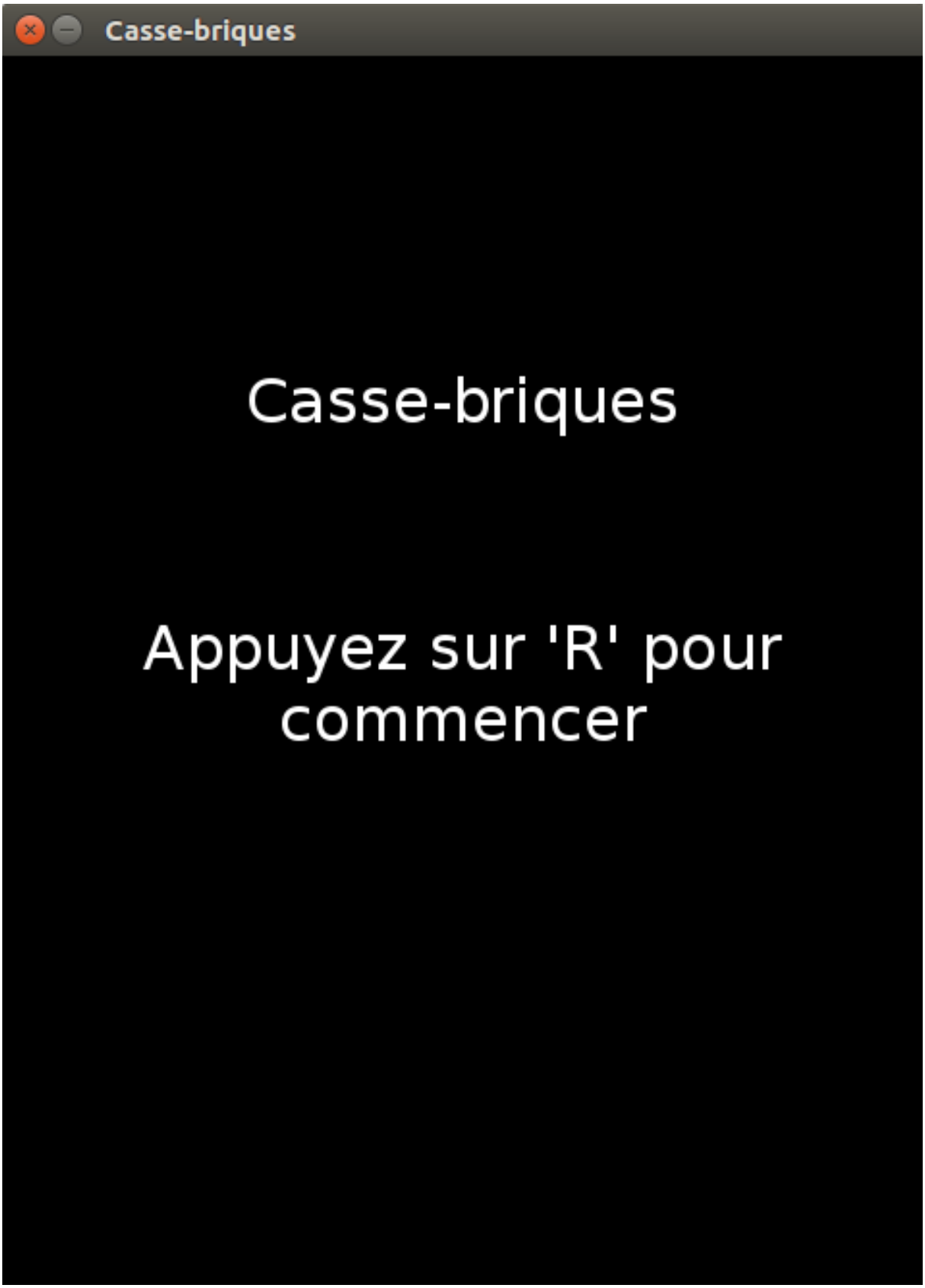

FIGURE 6. – La page du début.

Vous remarquerez que le second texte, prenant plus de taille en largeur que possible, est écrit automatiquement sur plusieurs lignes, comme convenu.

De plus, celui-ci indique qu'il faudra presser R pour lancer une partie. Par ailleurs, cette même touche permettra de redémarrer une partie. Ainsi, si nous sommes sur la page du début, nous avons juste à passer sur la page de jeu, sinon nous réinitialisons les variables. Ainsi, voici ce que nous allons ajouter à `love.keypressed` :

```
if key == "r" then
  if currentPage ~= PAGE_BEGINNING then

    resetRacket() -- Réinitialisation de la raquette

    -- Réinitialisation des briques
    for line=1, #bricks do
      for column=1, #bricks[line] do
        bricks[line][column].isNotBroken = true
      end
    end

    lives.count = NB_LIVES -- Réinitialisation des vies
    nbBricks = BRICKS_PER_COLUMN * BRICKS_PER_LINE --
        Réinitialisation du nombre de briques
    resetBall(racket.y) -- Réinitialisation de la balle

  end
  currentPage = PAGE_ROUND -- Page jeu
end
```

Comme vous pouvez le voir, nous réajustons juste les valeurs de nos tables au lieu d'en créer de nouvelles. Même s'il n'y a pas de soucis de performance à avoir au vu de la puissance des machines et des outils, je trouve plus propre de procéder ainsi.

Pour que ça fonctionne, il ne faut pas oublier de définir `resetRacket` qui permet de repositionner la raquette :

```
function resetRacket()
  racket.x = (WIN_WIDTH-racket.width) / 2 -- Position en abscisse
  racket.y = WIN_HEIGHT - 64 -- Position en ordonnée
end
```

Comme pour `resetBall` et `initializeBall`, nous avons ici un code dupliqué. Il est donc préférable de faire appel à `resetRacket` dans `initializeRacket`.

6.0.2. Quitter

Puisque jusqu'à présent nous sommes obligés de cliquer sur la croix rouge pour quitter le jeu, nous allons permettre de le faire en appuyant sur Échap. Comme vous vous en douter, nous allons donc ajouter quelques lignes à `love.keypressed` :

```lua
if key == "escape" then
  love.event.quit() -- Pour quitter le jeu
end
```

6.0.3. Page de fin

Pour le moment, notre jeu est fonctionnel, mais il ne se termine jamais. Nous allons donc faire en sorte qu'à la fin d'une partie, celui-ci s'arrête et qu'une page de fin s'affiche. Ainsi, nous allons tester l'état de la partie à la fin de notre `updateRound` et changer de page si besoin de cette manière :

```lua
if lives.count == 0 or nbBricks == 0 then
  currentPage = PAGE_END -- Page de fin
end
```

Maintenant nous n'avons plus qu'à afficher le contenu de la page de fin en fonction de si c'est une victoire ou une défaite dans `love.draw` :

```lua
local message = "Victoire !"
if lives.count == 0 then
  message = "Défaite !"
end
love.graphics.printf(message, 0, 0.25*WIN_HEIGHT, WIN_WIDTH,
    "center") -- Écriture
love.graphics.printf("Appuyez sur 'R' pour recommencer", 0,
    0.45*WIN_HEIGHT, WIN_WIDTH, "center") -- Écriture
```

Le résultat :

FIGURE 6. – La page de fin.

Voilà, notre jeu est désormais doté d'un joli menu !

7. Version finale et distribution

Notre casse-briques étant terminé, faisons un récapitulatif de l'organisation du projet, des sources ainsi que des améliorations possibles tout en voyant comment le distribuer :

7.0.1. Organisation et sources

```
casse_briques
    | images
        | --- icon.png
        | --- life.png
    | sounds
        | --- collision_brick.wav
        | --- collision_racket.wav
    | --- conf.lua
    | --- constants.lua
    | --- main.lua
```

Fichier **constants.lua**

Contenu masqué n°1

Fichier **conf.lua**

Contenu masqué n°2

Fichier **main.lua**

Contenu masqué n°3

7.0.2. Distribution

Maintenant que nous disposons d'une première version jouable de notre jeu, il peut être sympa d'en faire profiter les autres et de savoir ce qu'ils en pensent. En effet, quoi de plus normal que de vouloir que l'on joue à notre jeu et l'améliorer ? Avec Love2D, il existe plusieurs manières de distribuer son jeu.

7.0.2.1. En créant un fichier « .love »

La plus simple consiste à créer un fichier « **.love** » qui pourra être exécuté sur les machines ayant une version compatible de Love2D d'installée. Pour cela, il faut d'abord se placer dans le répertoire contenant tout notre jeu (code et ressources) et s'assurer que le fichier **main.lua** se trouve à la racine de celui-ci, sans quoi cela ne fonctionnera pas. Comme c'est bon dans notre cas, nous pouvons sélectionner tout le contenu puis le compresser en un fichier « **.zip** » (clique droit puis compresser sous *Ubuntu*). Une fois cela fait, il ne reste plus qu'à renommer le fichier en remplaçant l'extension par « *love* ». L'icône change et il est alors possible d'exécuter le jeu en double-cliquant sur le fichier.

7.0.2.2. En créant un fichier selon la plateforme visée

Pour distribuer notre jeu à des utilisateurs n'ayant pas forcément Love2D d'installé, il va falloir faire en fonction de la plateforme visée. Par exemple, il est possible de viser les utilisateurs de *Windows*, de *Mac OS X* ou encore d'*Android*. Nous ne traiterons que de la façon de faire un exécutable pour *Windows* dans ce tutoriel. Pour le reste, je vous laisse vous renseigner avec la documentation .

Pour faire un fichier « **.exe** », nous allons avoir besoin du fichier « **.love** » créé précédemment ainsi que de certains fichiers de Love2D. Ceux-ci sont téléchargeables sur la page principale , en prenant la version zippée 32 ou 64 bits.

Pour choisir la version adéquate à télécharger, sachez qu'un exécutable 32 bits pourra être exécuté sur une architecture 32 et 64 bits alors qu'un exécutable 64 bits ne sera utilisable qu'avec une architecture 64 bits. Ainsi, si vous souhaitez que votre exécutable soit utilisable sur n'importe laquelle de ces deux architectures, vous pouvez soit créer un exécutable 32 bits ou soit créer un exécutable pour chaque architecture.

Après avoir téléchargé le fichier compressé, Il nous faut en extraire le contenu puis copier-coller celui-ci dans un répertoire qui contiendra notre jeu exécutable. Ajoutons-y le fichier « **.love** » que l'on nommera **casse_briques.love** pour l'occasion, ainsi que les ressources. Ensuite, il ne reste plus qu'à créer l'exécutable à partir du fichier **casse_briques.love** et du **love.exe**. Sous *Linux* et *OS X*, nous entrerons `cat love.exe casse_briques.love > casse_briques.exe` dans le terminal tandis que sous *Windows* il faudra saisir `copy /b love.exe+casse_briques.love casse_briques.exe` en ligne de commandes (assurez-vous d'être dans le bon répertoire sinon ces commandes ne fonctionneront pas). En double-cliquant sur l'exécutable créé, le jeu doit se lancer. Si ce n'est pas le cas, assurez-vous que votre répertoire contienne à minima les fichiers suivants :

```
casse_briques_exe
    | images
        | --- icon.png
        | --- life.png
    | sounds
        | --- collision_brick.wav
        | --- collision_racket.wav
    | --- casse_briques.exe
    | --- license.txt (il est obligatoire d'inclure la licence de
        Love2D)
```

```
    | --- love.dll
    | --- lua51.dll
    | --- mpg123.dll
    | --- msvcp120.dll
    | --- msvcr120.dll
    | --- OpenAL32.dll
    | --- SDL2.dll
```

7.0.3. Améliorations possibles

Bien sûr, notre jeu pourrait être amélioré de multiples façons. Je vous donne ici quelques idées pour vous entraîner si vous le souhaitez.

Déjà, vous pouvez **réorganiser le code et le rendre plus modulable** par exemple en rajoutant des fonctions (une fonction pour dessiner les briques par exemple ou encore une autre pour gérer le déplacement de la raquette) ou en créant de nouvelles constantes (pour la vitesse de la raquette par exemple) ou encore en séparant le contenu de **main.lua** en plusieurs fichiers.

Ensuite, vous pouvez aussi essayer d'**ajouter des couleurs aux éléments** (en ajoutant une valeur couleur générée aléatoirement par exemple). Et puis, il serait possible de **rendre certaines briques plus résistantes que d'autres** (en rajoutant une valeur qui diminuerait de 1 à chaque collision avec la balle par exemple et une fois à 0 la brique serait considérée comme cassée). Plus compliqué, vous pourriez **faire que la balle soit ronde** (il faudrait alors dessiner un cercle et non plus un rectangle, ainsi que passer par une autre fonction de collision). Par ailleurs, il serait intéressant d'**ajouter des bonus** qui apparaîtraient lors de la destruction d'une brique et qu'il faudrait récupérer avec la raquette.

Enfin, sachez que des casse-briques fonctionnent en donnant à la balle non pas des vitesses de déplacement horizontal et vertical, mais directement un angle. Cela fait appel à des notions de trigonométrie et ça ne peut être qu'enrichissant à mettre en œuvre.

Bref, les idées ne manquent pas ! Si ça vous intéresse, vous avez donc de quoi vous occuper un moment.

8. Conclusion

Au terme de ce tutoriel, vous venez de découvrir Love2D et peut-être de réaliser votre premier jeu avec.

Sachez que nous avons seulement utilisé une infime partie de ce moteur. Si vous souhaitez en apprendre davantage, je vous renvoie à la documentation . Par ailleurs, si vous souhaitez pratiquer davantage, je vous recommande ce tutoriel .

À bientôt !

Merci à Gabbro pour ses retours et à nohar pour la validation.

Contenu masqué

Contenu masqué n°1

```lua
TITLE = "Casse-briques" -- Titre
PATH_ICON = "images/icon.png" -- Chemin image icône
WIN_WIDTH = 480 -- Largeur fenêtre
WIN_HEIGHT = 640 -- Hauteur fenêtre

BRICKS_PER_LINE = 7 -- Nombre de briques par ligne
BRICKS_PER_COLUMN = 6 -- Nombre de briques par colonne

NB_LIVES = 3 -- Nombre de vies initiales
PATH_LIFE = "images/life.png" -- Chemin image vie

DEFAULT_SPEED_BX = 130 -- Vitesse horizontale
DEFAULT_SPEED_BY = 335 -- Vitesse verticale

PATH_SOUND_BRICK = "sounds/collision_brick.wav" -- Chemin son brique
PATH_SOUND_RACKET = "sounds/collision_racket.wav" -- Chemin son raquette

PAGE_BEGINNING = 1 -- Page de début
PAGE_ROUND = 2 -- Page de partie
PAGE_END = 3 -- Page de fin

-- Fonction pour tester la collision entre deux rectangles
function collideRect(rect1, rect2)
  if rect1.x < rect2.x + rect2.width and
     rect1.x + rect1.width > rect2.x and
     rect1.y < rect2.y + rect2.height and
     rect1.height + rect1.y > rect2.y then
       return true
  end
  return false
end
```

Retourner au texte.

Contenu masqué n°2

```lua
require('constants')

function love.conf(t)

  t.window.title = TITLE -- Change le titre de la fenêtre
  t.window.icon = PATH_ICON -- Change l'icone de la fenêtre
  t.window.width = WIN_WIDTH -- Change la largeur de la fenêtre
  t.window.height = WIN_HEIGHT -- Change la hauteur de la fenêtre

end
```

Retourner au texte.

Contenu masqué n°3

```lua
require('constants')

-- pour écrire dans la console au fur et à mesure, facilitant ainsi le débogage
io.stdout:setvbuf('no')

-- [[ Variables locales ]]

local racket -- Déclaration variable pour la raquette
local bricks -- Déclaration variable pour les briques
local lives -- Déclaration variable pour les vies
local ball -- Déclaration variable pour la balle

local nbBricks = BRICKS_PER_COLUMN * BRICKS_PER_LINE -- Nombre de briques
local currentPage = PAGE_BEGINNING -- Page courante

local soundBrick -- Déclaration variable son brique
local soundRacket -- Déclaration variable son raquette
local font -- Déclaration variable pour la police d'écriture

--[[ Fonctions raquette ]]

function initializeRacket()

  racket = {} -- Initialisation variable pour la raquette

  -- Initialisation de paires (clef, valeur) de la table racket
```

```
27    racket.speedX = 215 -- Vitesse horizontale
28    racket.width = WIN_WIDTH / 4 -- Largeur
29    racket.height = WIN_HEIGHT / 37 -- Hauteur
30    resetRacket() -- Position
31
32  end
33
34
35  function resetRacket()
36
37    racket.x = (WIN_WIDTH-racket.width) / 2 -- Position en abscisse
38    racket.y = WIN_HEIGHT - 64 -- Position en ordonnée
39
40  end
41
42  -- [[ Fonctions briques ]]
43
44  function createBrick(line, column)
45
46    -- Fonction pour créer une brique et l'initialiser en fonction de
          sa position dans le mur
47    local brick = {}
48    brick.isNotBroken = true -- Brique pas encore cassée
49    brick.width = WIN_WIDTH / BRICKS_PER_LINE - 5 -- Largeur
50    brick.height = WIN_HEIGHT / 35 -- Hauteur
51    brick.x = 2.5 + (column-1) * (5+brick.width) -- Position en
          abscisse
52    brick.y = line * (WIN_HEIGHT/35+2.5) -- Position en ordonnée
53    return brick
54
55  end
56
57  function initializeBricks()
58
59      bricks = {} -- Initialisation variable pour les briques
60      for line=1, BRICKS_PER_COLUMN do
61        table.insert(bricks, {}) -- Ajout d'une ligne
62        for column=1, BRICKS_PER_LINE do
63          local brick = createBrick(line, column)
64          table.insert(bricks[line], brick) -- Ajout d'une brique par
                colonne de la ligne
65        end
66      end
67
68  end
69
70  -- [[ Fonction vie ]]
71
72  function initializeLives()
73
```

Contenu masqué

```lua
  lives = {} -- Initialisation variable pour les vies
  lives.count = NB_LIVES -- Nombre de vie
  lives.img = love.graphics.newImage(PATH_LIFE) -- Image vie
  lives.width, lives.height = lives.img:getDimensions() --
      Dimensions de l'image

end

-- [[ Fonctions balle ]]

function initializeBall(racketHeight, racketY)

  ball = {} -- Initialisation variable pour la balle
  ball.width, ball.height = racketHeight * 0.75, racketHeight *
      0.75  -- Taille
  resetBall(racketY)

end


function resetBall(racketY)

  ball.speedY = -DEFAULT_SPEED_BY -- Vitesse verticale
  ball.speedX = math.random(-DEFAULT_SPEED_BX, DEFAULT_SPEED_BX) --
      Vitesse horizontale
  ball.x = WIN_WIDTH / 2 - ball.width / 2 -- Position en abscisse
  ball.y = racketY - 2 * ball.height - ball.height / 2 -- Position
      en ordonnée

end


function collisionBallWithRacket()

    soundRacket:play() -- Joue le son raquette

    -- Collision par la gauche (coin haut inclus)
    if ball.x < racket.x + 1/8 * racket.width and ball.speedX >= 0
        then
      if ball.speedX <= DEFAULT_SPEED_BX/2 then -- Si vitesse trop
          faible
        ball.speedX = -math.random(0.75*DEFAULT_SPEED_BX,
            DEFAULT_SPEED_BX) -- Nouvelle vitesse
      else
        ball.speedX = -ball.speedX
      end
    -- Collision par la droite (coin haut inclus)
    elseif ball.x > racket.x + 7/8 * racket.width and ball.speedX
        <= 0 then
```

```
115       if ball.speedX >= -DEFAULT_SPEED_BX/2 then  -- Si vitesse
              trop faible
116         ball.speedX = math.random(0.75*DEFAULT_SPEED_BX,
                DEFAULT_SPEED_BX) -- Nouvelle vitesse
117       else
118         ball.speedX = -ball.speedX
119       end
120     end
121     -- Collision par le haut
122     if ball.y < racket.y and ball.speedY > 0 then
123       ball.speedY = -ball.speedY
124   end
125
126 end
127
128
129 function collisionBallWithBrick(ball, brick)
130
131   soundBrick:play() -- Joue le son brique
132
133   -- Collision côté gauche brique
134   if ball.x < brick.x and ball.speedX > 0 then
135       ball.speedX = -ball.speedX
136   -- Collision côté droit brique
137   elseif ball.x > brick.x + brick.width and ball.speedX < 0 then
138       ball.speedX = -ball.speedX
139   end
140   -- collision haut brique
141   if ball.y < brick.y and ball.speedY > 0 then
142     ball.speedY = -ball.speedY
143   -- Collision bas brique
144   elseif ball.y > brick.y and ball.speedY < 0 then
145     ball.speedY = -ball.speedY
146   end
147
148   brick.isNotBroken = false -- Brique maintenant cassée
149   nbBricks = nbBricks - 1 -- Ne pas oublier de décrémenter le
          nombre de briques
150
151 end
152
153 -- [[ Fonctions de la page partie ]]
154
155 function updateRound(dt)
156
157   -- Mouvement vers la gauche
158   if love.keyboard.isDown('left', 'q') and racket.x > 0 then
159       racket.x = racket.x - (racket.speedX*dt)
160   -- Mouvement vers la droite
```

Contenu masqué

```
161   elseif love.keyboard.isDown('right', 'd') and racket.x +
          racket.width < WIN_WIDTH then
162       racket.x = racket.x + (racket.speedX*dt)
163   end
164
165   -- Mise à jour position de la balle
166   ball.x = ball.x + ball.speedX * dt -- Position en abscisse
167   ball.y = ball.y + ball.speedY * dt -- Position en ordonnée
168
169   -- Collision de la balle avec la fenêtre
170   if ball.x + ball.width >= WIN_WIDTH then  -- Bordure droite
171     ball.speedX = -ball.speedX
172   elseif ball.x <= 0 then -- Bordure gauche
173     ball.speedX = -ball.speedX
174   end
175   if ball.y <= 0 then  -- Bordure haut
176     ball.speedY = -ball.speedY
177   elseif ball.y + ball.height >= WIN_HEIGHT then -- Bordure bas
178     lives.count = lives.count - 1
179     resetBall(racket.y)
180   end
181
182   -- Collision entre la balle et la raquette
183   if collideRect(ball, racket) then
184     collisionBallWithRacket()
185   end
186
187   -- Collision de la balle avec les briques
188   for line=#bricks, 1, -1 do
189     for column=#bricks[line], 1, -1 do
190       if bricks[line][column].isNotBroken and collideRect(ball,
              bricks[line][column]) then
191         collisionBallWithBrick(ball, bricks[line][column]) --
                Collision entre la balle et une brique
192       end
193     end
194   end
195
196   -- Test de l'état de la partie
197   if lives.count == 0 or nbBricks == 0 then
198     currentPage = PAGE_END -- Page de fin
199   end
200 end
201
202
203 function drawRound()
204
205   love.graphics.setColor(255, 255, 255) -- Couleur blanche
206
207   -- Affichage de raquette
```

```lua
  love.graphics.rectangle('fill', racket.x, racket.y, racket.width, racket.height)

  -- Affichage des briques
  for line=1, #bricks do -- Ligne
    for column=1, #bricks[line] do -- Colonne
      local brick = bricks[line][column]
      if brick.isNotBroken then -- Si la brique n'est pas cassée
        love.graphics.rectangle('fill', brick.x, brick.y, brick.width, brick.height)
      end
    end
  end

  -- Affichage des vies
  for i=0, lives.count-1 do -- Pour chaque vie
    local posX = 5 + i * 1.20 * lives.width -- Calcul de la position en abscisse
    love.graphics.draw(lives.img, posX, WIN_HEIGHT-lives.height) -- Affichage de l'image
  end

  -- Affichage de la balle
  love.graphics.rectangle('fill', ball.x, ball.y, ball.width, ball.height)

end

-- [[ Fonctions Callback de Love2D ]]

function love.load()

  math.randomseed(love.timer.getTime()) -- Initialisation de la graine avec un temps en ms

  initializeRacket()
  initializeBricks()
  initializeLives()
  initializeBall(racket.height, racket.y)

  soundBrick = love.audio.newSource(PATH_SOUND_BRICK, "static") -- Chargement son brique
  soundRacket = love.audio.newSource(PATH_SOUND_RACKET, "static") -- Chargement son raquette

  font = love.graphics.newFont(32) -- Initialise font avec une police de taille 32
  love.graphics.setFont(font) -- Définit font comme la police utilisée

```

```
248 end
249
250
251 function love.update(dt)
252
253   if currentPage == PAGE_BEGINNING then
254     -- Traitement page début
255   elseif currentPage == PAGE_ROUND then
256     updateRound(dt)
257   elseif currentPage == PAGE_END then
258     -- Traitement page fin
259   end
260
261 end
262
263 function love.draw()
264
265   if currentPage == PAGE_BEGINNING then
266
267     love.graphics.printf("Casse-briques", 0, 0.25*WIN_HEIGHT,
          WIN_WIDTH, "center") -- Écriture
268     love.graphics.printf("Appuyez sur 'R' pour commencer", 0,
          0.45*WIN_HEIGHT, WIN_WIDTH, "center") -- Écriture
269
270   elseif currentPage == PAGE_ROUND then
271
272     drawRound()
273
274   elseif currentPage == PAGE_END then
275
276     local message = "Victoire !"
277     if lives.count == 0 then
278       message = "Défaite !"
279     end
280     love.graphics.printf(message, 0, 0.25*WIN_HEIGHT, WIN_WIDTH,
          "center") -- Écriture
281     love.graphics.printf("Appuyez sur 'R' pour recommencer", 0,
          0.45*WIN_HEIGHT, WIN_WIDTH, "center") -- Écriture
282
283   end
284
285 end
286
287
288 function love.keypressed(key)
289
290   if key == "r" then
291     if currentPage ~= PAGE_BEGINNING then
292
293       resetRacket() -- Réinitialisation de la raquette
```

Contenu masqué

```
294
295       -- Réinitialisation des briques
296       for line=1, #bricks do
297         for column=1, #bricks[line] do
298           bricks[line][column].isNotBroken = true
299         end
300       end
301
302       lives.count = NB_LIVES -- Réinitialisation des vies
303       nbBricks = BRICKS_PER_COLUMN * BRICKS_PER_LINE --
              Réinitialisation du nombre de briques
304       resetBall(racket.y) -- Réinitialisation de la balle
305
306     end
307     currentPage = PAGE_ROUND -- Page jeu
308
309   end
310
311   if key == "escape" then
312     love.event.quit() -- Pour quitter le jeu
313   end
314
315 end
```

Retourner au texte.