AzurionQuest

Documentation technique

BlackCrew

BlackCrew est le nom de notre équipe de projet, composée de **Julien Sergent, Mathieu Vandeginste, Florian Faity & Quentin Pomarel**

Développement

Plus de 220 heures de développement ont été nécessaires pour coder ce jeu vidéo qui mêle le grand classique de l'Arkanoid au thème de la science-fiction.

SDL

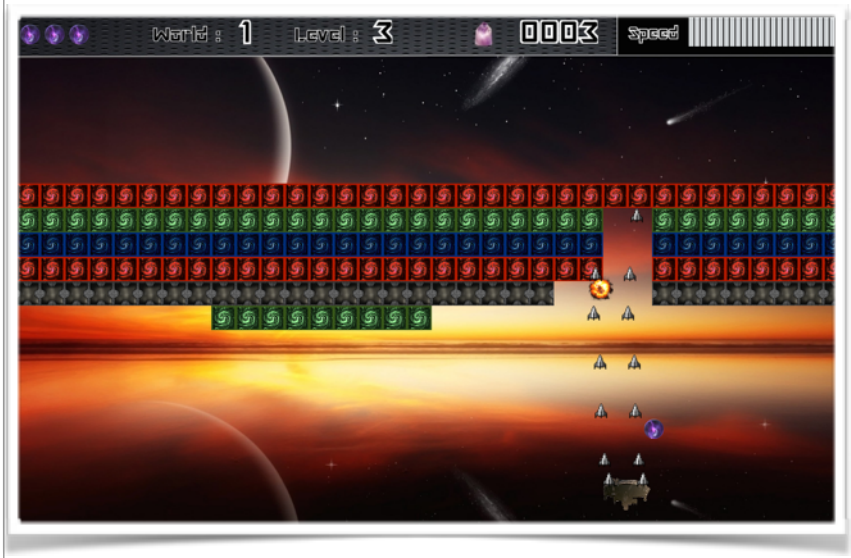
Le SDL est la librairie C qui nous a permis de développer ce jeu. Notre volonté a été dès le début de pousser les graphismes dans le thème de la science-fiction au plus haut niveau dans un jeu totalement immersif en plein écran.

D'autres librairies annexes ont été utilisées afin de gérer les formats d'images (SDL_Image), les sons et musiques (FMODEX)

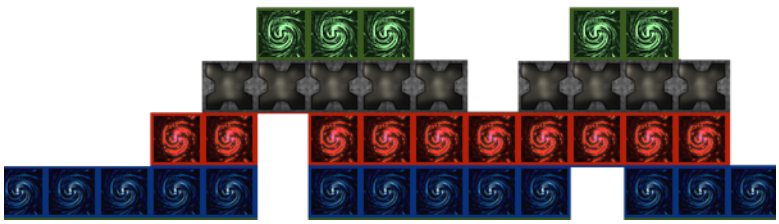
Introduction

Cette documentation technique explique nos choix d'implémentation, et montre tous les détails à propos :

- *de la structure de notre jeu*
- *du moteur d'éjection du paddle (angles)*
- *du moteur de collisions*
- *de la gestion des bonus/malus*



Ce document n'est pas un document commercial et s'adresse aux examinateurs de notre projet Supinfo.



Sommaire

Une première !

En tant qu'étudiants de première année à Supinfo, et bien qu'il ne s'agisse pas de notre première expérience de développement en C, il s'agit pour nous de notre projet jeu à interface graphique.

Il nous a fallu plus de deux mois avant de nous lancer dans le développement du jeu à proprement parler car il s'agissait d'abord de comprendre la librairie SDL en lisant des tutoriaux (notamment sur le site openclassrooms.com) et en réalisant des TP proposés (Mario Sokoban)

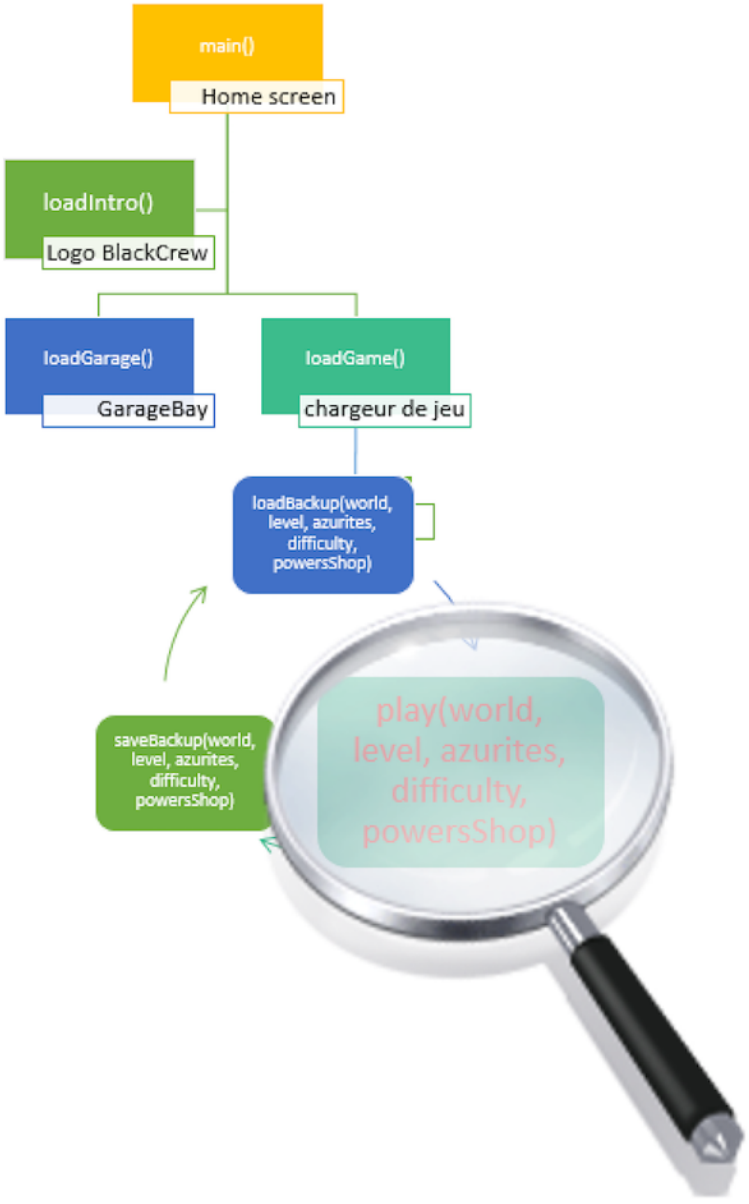
Difficultés

Bien que le C soit un langage extrêmement performant car de bas-niveau et bien que nous ayons eu une excellente expérience de développement, nous pourrions dire avec du recul qu'il aurait été plus simple de se contenter d'un site qui propose soit le jeu en téléchargement, soit le jeu en ligne codé avec des technologies web comme JavaScript. En effet la technologie de portage du C, NaCl de Google, n'a pas été massivement adoptée, est très lourde à mettre en place et uniquement compatible avec le navigateur Google Chrome.

1. Structure du jeu

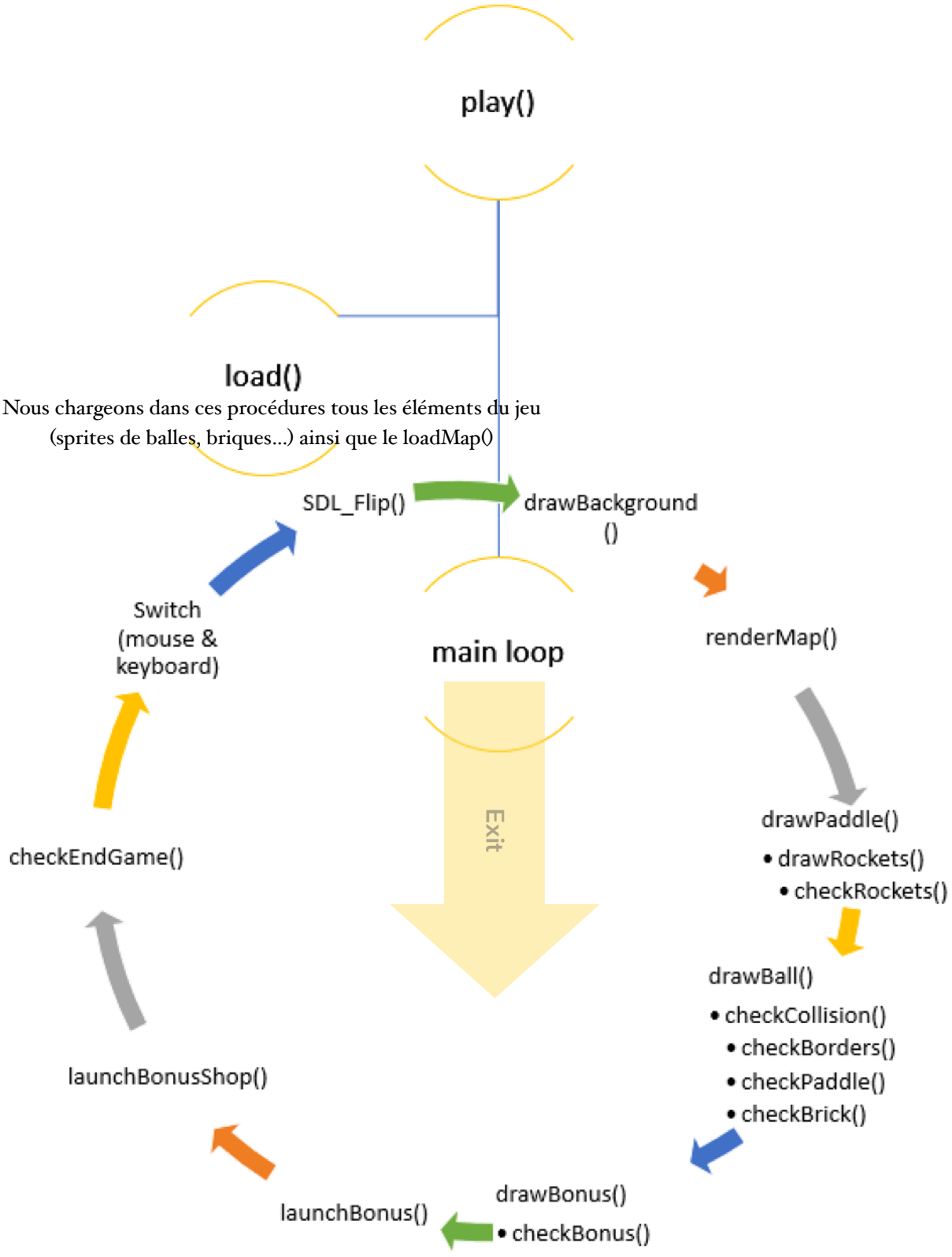
Afin de faciliter le développement du jeu et son évolutivité, nous avons intégralement structuré le jeu en différents fichiers (.cpp / .h) et fonctions, commenté tout notre code et nommé nos variables et de nombreuses énumérations en anglais afin d'être compréhensible par n'importe quel développeur.

Notre programme comporte 42 fonctions/procédures et plus de 3000 lignes de codes.



Voici un schéma simplifié de la structure de notre jeu.

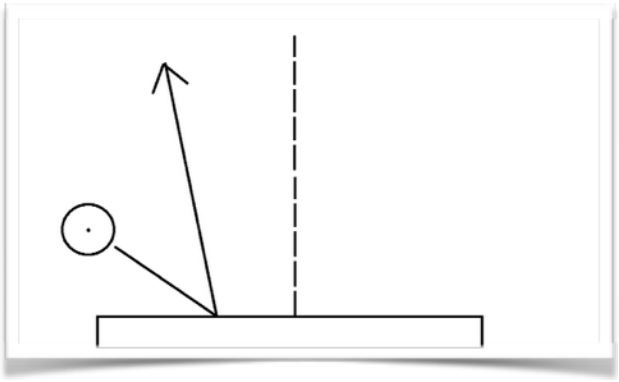
Rentrons en détail dans la procédure play (fichier game.cpp) :

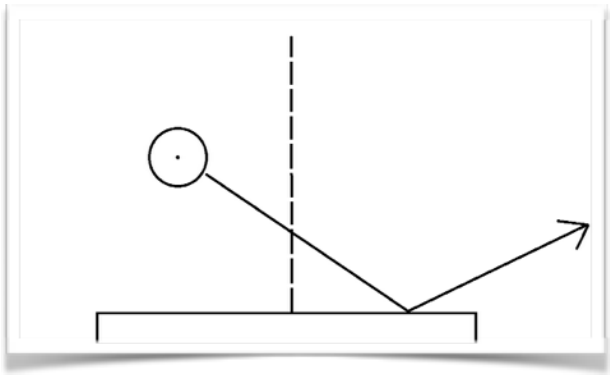


2.Moteur d'éjection du paddle (angles)

Le moteur d'éjection du paddle affecte un angle de trajectoire à la balle en fonction de la position x de celle-ci, grâce à deux fonctions affines.

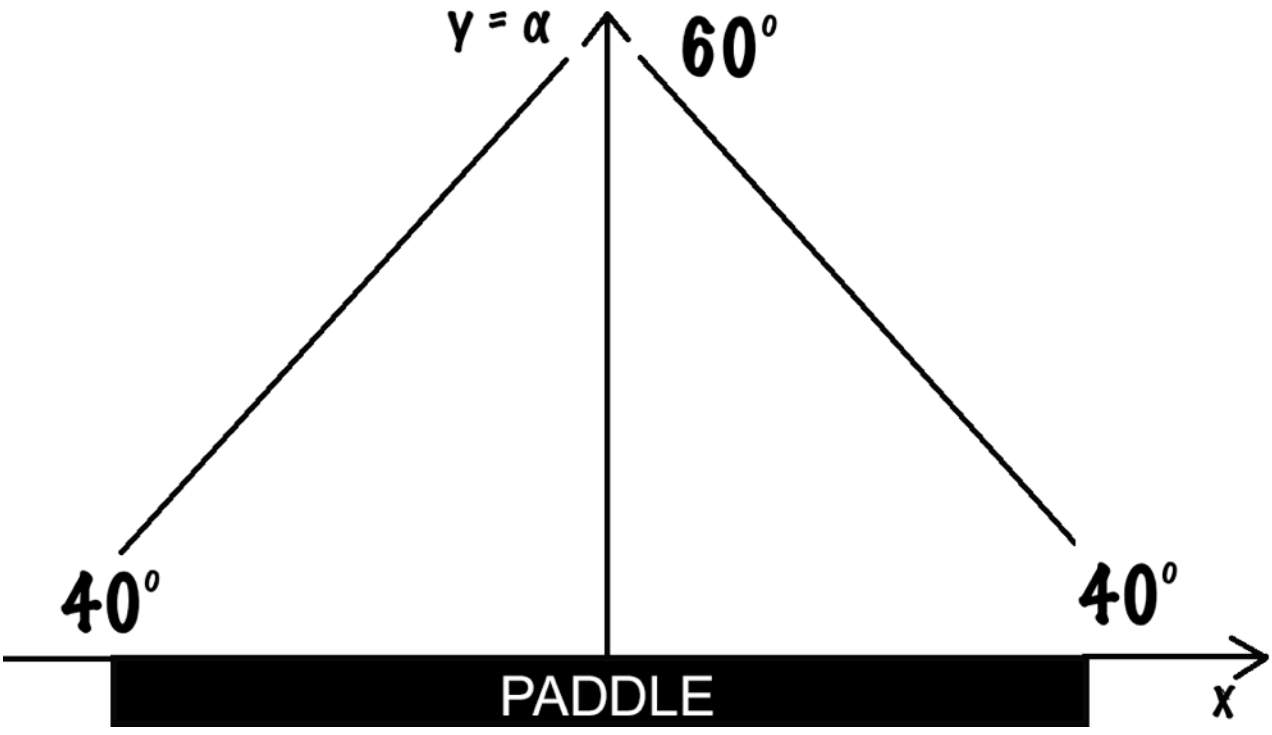
```
if( ( angleBall[BL][X] + RADIUS ) <= (componentsPaddle[X] + componentsPaddle[HALFSIZE]) ){
    componentsBall[DIRX] = -componentsBall[DIRX];
    componentsBall[DIRY] = -componentsBall[DIRY];
    componentsBall[ANGLE] = ( ( 20 * ( angleBall[BR][X] - componentsPaddle[X] ) ) / componentsPaddle[HALFSIZE] ) + 40;
}else{
    componentsBall[ANGLE] = ( ( 20 * ( componentsPaddle[X] - angleBall[BL][X] ) ) / componentsPaddle[HALFSIZE] ) + 80;
    componentsBall[DIRY] = -componentsBall[DIRY];
}
```





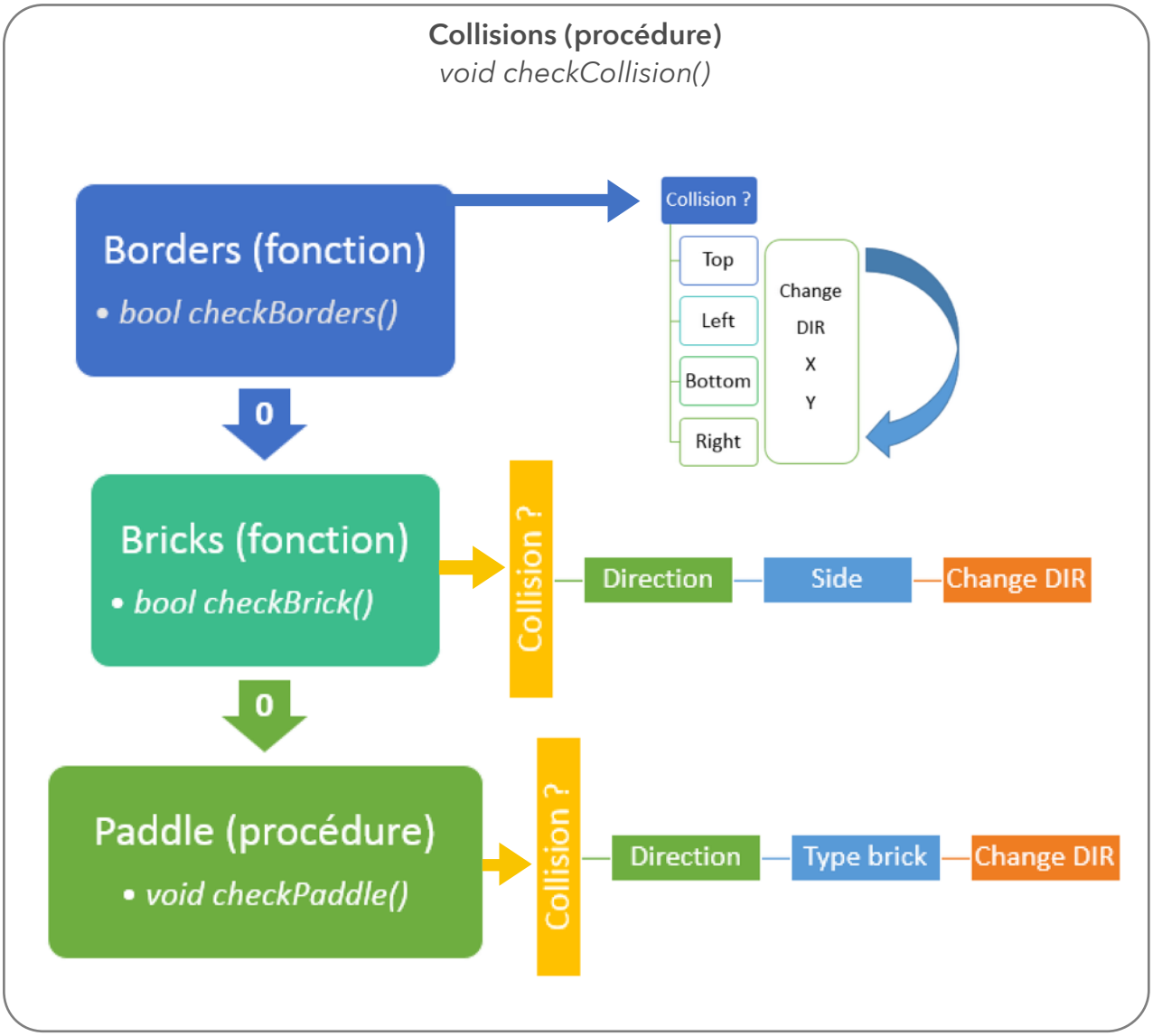
Le but est de renvoyer la balle :

- systématiquement verticalement
- et horizontalement si elle touche le paddle dans sa première moitié en fonction de sa provenance
- avec un angle de réflexion qui est fonction du lieu d'impact sur le paddle.



3.Moteur de collisions général

Comme il est expliqué dans la partie 1 (le schéma général du programme), les collisions sont gérées dans la procédure checkCollision() qui appelle trois sous-fonctions/procédures :

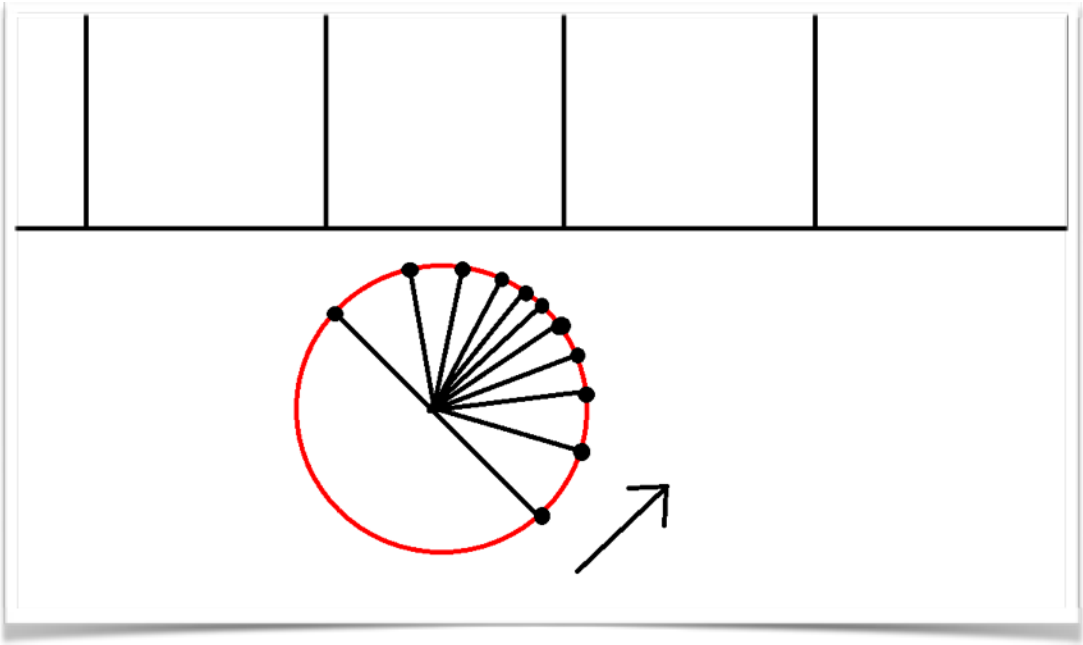


Le chapitre suivant explique en détails la fonction checkBrick.

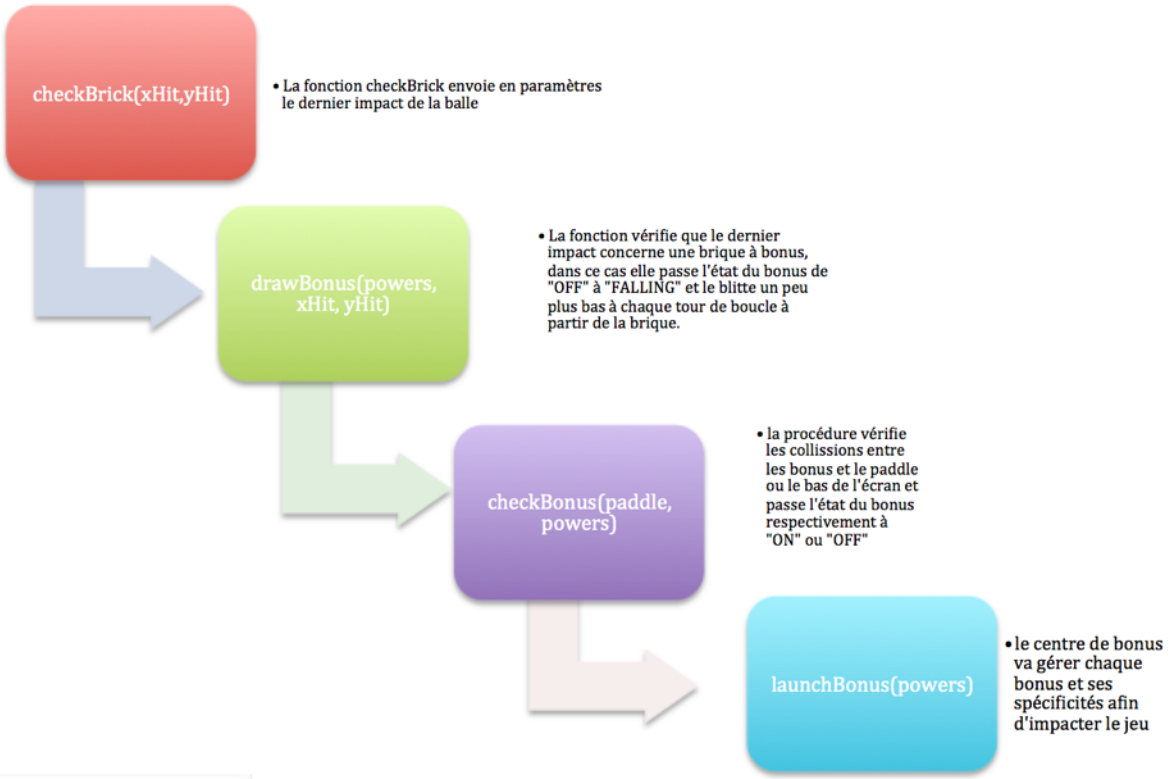
4.Moteur de collisions des briques

Afin de parvenir à un moteur de collisions parfaitement stable, nous avons dû repenser à plusieurs reprises car nous avions des problèmes rencontrés à plusieurs échelles :

- 1ère version : nous résonnions par rapport à l'origine de la balle qui est une image carrée et ajoutions des marges en fonction de la direction x et y, de nombreux problèmes dont traversées du métal;
- 2ème version : deux angles gérés mais toujours de nombreuses collisions non prises en compte;
- 3ème version : 4 angles gérés (top/bottom/left/right) : beaucoup moins de traversées mais toujours des «sauts de briques»;
- 4ème version : nous avons remis à plat tout notre raisonnement et notre algorithme en vérifiant à chaque pixel une éventuelle collision : il s'agit de raisonner avec des points de la demi-périphérie de la balle en fonction de sa direction (calculs trigonométriques). Nous raisonnions sur 7 points en V4.0, puis 9 points en V4.0.1. Les collisions étaient presque parfaites mais dans de très rares cas un saut de brique se produisait. La version 4.1 gère tous les points de (cos ($\alpha+\pi/2$) , sin($\alpha+\pi/2$)) à (cos ($\alpha-\pi/2$) à sin ($\alpha-\pi/2$)) et a grandement simplifié les versions 4 précédentes par une généralisation de ces points en une boucle.



5.Gestion des bonus & malus



Pour finir, un exemple concret : le cas du bonus « Rockets »