Space Buster: de la conception à la réalisation d'un jeu vidéo en C#

Travail de maturité réalisé au Lycée Denis-de-Rougemont de Neuchâtel
sous la direction de M. Jean-Marc Ledermann

Radu Cotofrei

Avant-Propos

Idée générale

Le but de ce travail de maturité est de développer un jeu vidéo basé sur le célèbre concept du « casse-briques », en incluant tout l'univers qui gravite autour du jeu en lui-même. Afin de rentrer dans le vif du sujet, je vais commencer par donner quelques informations sur ma motivation et sur les jeux vidéo.

Choix du sujet

Je suis passionné aussi bien d'informatique que de jeux vidéo. J'aime programmer car cela me permet de m'exprimer et d'avoir la sensation de pouvoir créer un monde qui m'appartient. Étant donné que je souhaite par la suite devenir développeur dans ce monde vidéo-ludique, il m'a paru opportun de me lancer dans la création d'un jeu vidéo.

Tout au long de mon travail, j'ai essayé de reproduire aussi fidèlement que possible un jeu dit à grand budget (les triples A), autour duquel tourne une véritable entreprise marketing. Le but de ce travail de maturité n'est donc pas seulement de créer un jeu, mais aussi de le faire vivre. Comme tous les grands jeux, il possède une boîte avec une jaquette dédiée, une notice d'utilisation et un site web (Site officiel de Space Buster, 2012). Tout cet univers autour du produit en lui-même contribue à lui donner de la valeur et à l'enrichir.

J'ai également développé un éditeur qui permet de créer des niveaux de jeu de façon simple et rapide. Sa conception et son utilisation font l'objet d'un chapitre entier de ce document.

But d'un jeu vidéo

Le jeu vidéo est un monde virtuel où tout est possible. Cet art, si l'on peut utiliser ce mot, permet à l'imagination et à l'inspiration de leurs disciples de prendre forme et de les faire partager grâce à l'informatique et l'infographie.

Le but premier d'un jeu vidéo est de divertir les joueurs et de les transporter dans un autre univers. Les meilleurs jeux sont ceux qui possèdent une grande immersion et permettent facilement aux joueurs de s'identifier au monde virtuel.

Le concept du jeu réalisé dans le cadre de ce travail

Le jeu développé s'appelle Space Buster. Le jeu reprend le concept du « casse-briques ». Le principe général est de détruire, au moyen d'une balle ou d'un rayon laser, un ensemble de cibles se trouvant dans un niveau jeu pour accéder au niveau suivant. Le joueur contrôle un vaisseau muni d'une barre qu'il peut déplacer sur un axe horizontal (et partiellement vertical). Le but est d'empêcher la balle de franchir la partie inférieure de la fenêtre du jeu en la touchant avec la barre. S'il y parvient, la balle est renvoyée en direction opposée ; dans le cas contraire, le joueur perd la balle et une vie. S'il n'a plus de vie, il perd la partie. Toute la difficulté est donc de rattraper la balle lorsqu'elle se déplace vite, tout en évitant les différentes cibles qui descendent.

Les outils de création

Initialement, j'avais prévu de développer le jeu en JavaScript. En effet, j'avais déjà développé une version basique en cours d'informatique, ce qui me permettait de partir d'une bonne base. Cependant, le langage JavaScript, même s'il reste un langage facile à appréhender, n'en demeure pas moins limité pour un jeu vidéo. Il est vrai que de nombreux jeux sur internet sont codés en JavaScript, ou en Flash, mais leur concept est simpliste et leur réalisation sommaire. Si le jeu devient trop complexe, il commence à saccader. C'est pourquoi j'ai décidé de me tourner vers un langage plus adapté à un jeu

vidéo, un langage développé par Microsoft : le C Sharp (C#). Ce dernier est beaucoup plus puissant, notamment grâce aux bibliothèques fournies par Microsoft qui contiennent du code permettant d'avoir accès à bon nombre de fonctions très utiles. Sa puissance de calcul est également plus élevée que celle de JavaScript.

Autre avantage considérable du langage de Microsoft, par rapport au Java de Sun, est l'existence de l'environnement XNA, qui permet le développement de jeux vidéo sur PC, Xbox 360 ou Windows Phone. XNA inclut une série de classes et de méthodes qui simplifient le développement d'un jeu car la logique du jeu est directement implantée dans le modèle de base (en anglais, « template »).

Impressions laissées aux joueurs

Le but d'un jeu, comme expliqué plus haut, est de transporter le joueur dans un autre univers et de lui permettre d'accomplir des choses qu'il ne serait pas capable de faire en temps normal. Il faut que l'univers du jeu soit autant cohérent que possible pour que le joueur se sente immergé et ne voit plus le jeu comme une simple reproduction du monde réel. Si le joueur est captivé par le jeu et a envie d'aller jusqu'au bout, alors le jeu est un succès.

Sources d'inspiration

Étant un joueur depuis bon nombre d'années déjà, j'ai pu essayer une grande variété de jeux et je peux dire que chaque jeu m'a donné une petite inspiration. En effet, en parcourant un jeu, je me rends compte des points positifs et négatifs, ce qui me permet par la suite de juger un jeu à sa juste valeur, et également de développer des jeux que je juge intéressants.

Comme dit plus haut, le concept de mon jeu s'inspire en grande partie des jeux de « casse-briques ». Pour apporter une nouveauté, j'ai ajouté un mouvement aux cibles. Elles ne sont plus immobiles mais descendent vers le vaisseau. Le tout devient plus dynamique et on est moins susceptible de s'ennuyer.

Les outils nécessaires

L'espace de développement : Microsoft Visual Studio C# Express

Visual C# est un outil de développement édité par Microsoft, permettant de concevoir des applications articulées autour du langage C#. Visual C# propose les outils pour développer des applications C# hautement performantes qui ciblent la plateforme nouvelle génération de Microsoft pour la programmation distribuée et compatible Internet. Ce langage de programmation est simple, de type sécurisé et orienté objet. Il a été conçu pour générer des applications d'entreprise. Le code écrit en C# est compilé en code managé[1] *exécuté sous le Framework .NET.* (Wikipédia, Visual C Sharp, 2009)

La version Express a la particularité d'être gratuite. Elle est amplement suffisante pour créer le jeu[2].

Visual Studio permet aussi de déceler rapidement une faute ou une erreur dans le code, avant même de compiler (en anglais, « build ») le projet. Il est également très ergonomique visuellement et on accède rapidement aux fonctions cherchées. Son seul petit inconvénient réside dans sa sauvegarde automatique après chaque compilation. Il aurait été plus judicieux de sauvegarder au moment de la fermeture de Visual Studio, car si on se rend compte que les modifications apportées à notre projet ne nous conviennent pas la seule option pour revenir à la version antérieure est de remodifier à nouveau le code.

Le langage C#

Le C# (c Sharp) est un langage de programmation orienté objet à typage fort[3]*, créé par la société Microsoft, et notamment par un de ses employés, Anders Hejlsberg, le créateur du langage Delphi.*

[1] Le managed code (code managé), terme spécifique à Microsoft, est un code source qui s'exécute sous le contrôle de la machine virtuelle CLR (Common Language Runtime). Ce terme est employé par opposition au unmanaged code (code non géré), qui est exécuté directement par le processeur.

[2] Elle est disponible à l'adresse: http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express

[3] Un langage de programmation est dit fortement typé lorsqu'il garantit que les types de données employés décrivent correctement les données manipulées (Wikipedia). La compilation ou l'exécution peuvent détecter des erreurs de typage.

Il a été créé afin que la plate-forme Microsoft .NET soit dotée d'un langage permettant d'utiliser toutes ses capacités. Il est très proche du Java dont il reprend la syntaxe générale ainsi que les concepts (la syntaxe reste cependant relativement semblable à celles de langages tels que le C++ et le C). Un ajout notable à Java est la possibilité de surcharge des opérateurs, inspirée du C++. Toutefois, l'implémentation de la redéfinition est plus proche de celle du Pascal Objet. (Wikipédia, C Sharp, 2012)

Lorsqu'on dit que le C Sharp est un langage orienté objet, cela signifie qu'il utilise des objets (des sous-ensembles de code autonomes) qui peuvent communiquer entre eux. Par exemple, il suffit de définir l'objet « brique » une fois, puis de l'instancier autant de fois que l'on veut. L'objet représente la définition, le concept de quelque chose et l'instance représente sa concrétisation, son utilisation pratique.

J'ai commencé à apprendre le C# pendant mes vacances d'été 2011, grâce notamment à un site web très pratique : Le Site du Zéro[1]. Malgré mes connaissances encore limitées du langage, je peux tout de même donner un premier avis. Je trouve personnellement ce langage très agréable comparé aux autres langages que j'ai pu essayer (Visual Basic 6, Visual Basic.NET, JavaScript, PHP, C). Quand je code en C#, j'ai l'impression d'avoir une grande puissance à ma disposition et de pouvoir créer tout ce que je désire, grâce à toutes les bibliothèques disponibles. Pour ce qui est de la syntaxe, je la trouve claire et logique.

Microsoft XNA

Microsoft XNA, officiellement XNA's Not Acronymed, parfois présenté dans les médias Xbox Next-Generation Architecture, désigne une série d'outils fournis gratuitement par Microsoft qui facilite les développements de jeux pour les plates-formes Windows, Zune, Windows Phone 7, et Xbox 360 en réunissant un maximum d'outils en provenance de Microsoft et de ses partenaires...

Il contient principalement un framework, des outils d'intégrations de contenu, et la documentation nécessaire. L'interface de développement (IDE) utilisé, à télécharger séparément, est Visual Studio.

Avec XNA, Microsoft est le premier constructeur à ouvrir la porte au développement indépendant sur sa console Xbox 360. Les jeux produits sont distribués via le Xbox Live. (Wikipedia, Microsoft, XNA).

Pour avoir plus d'informations, on peut consulter le site consacré au développement des applications et des jeux par Microsoft[2]. Le site est très utile car il propose des tutoriels et des aides. Il permet également des télécharger des projets-démo pour essayer plusieurs fonctions de XNA.

Avant le codage du jeu

La conception

Avant de commencer à coder, il faut bien avoir en tête le déroulement complet du jeu, du moment où on le lance jusqu'à ce qu'il ferme. La meilleure approche consiste à créer un schéma simplifié mais complet, qui montre comment le jeu est censé se dérouler.

La logique

Pour ce qui est de la logique, Microsoft a implémenté une logique de jeu dans XNA. On commence par initialiser le jeu, on attribue des valeurs aux variables, puis on charge les données du jeu dans le projet. Le cœur du jeu, c'est une boucle infinie où seront répétés une soixantaine de fois par seconde les procédures de mise à jour et d'affichage. Cette logique est la base sur laquelle nous allons construire le jeu. Il faut bien avoir en tête que cette logique est appliquée à n'importe quelle classe. Pour gérer toutes ces classes et permettre d'avoir un menu, des options et un jeu principal, il faut créer un gestionnaire d'écran qui va contrôler l'affichage des différents écrans. Pour cela, Microsoft met à disposition une classe nommée *ScreenManager* (Game State Management, 2011) (ainsi que d'autres classes complémentaires).

Schéma

Le schéma représenté dans la figure 1 permet de se faire une idée de ce que fait le jeu et à quel moment.

[1] http://www.siteduzero.com
[2] http://create.msdn.com/en-US/

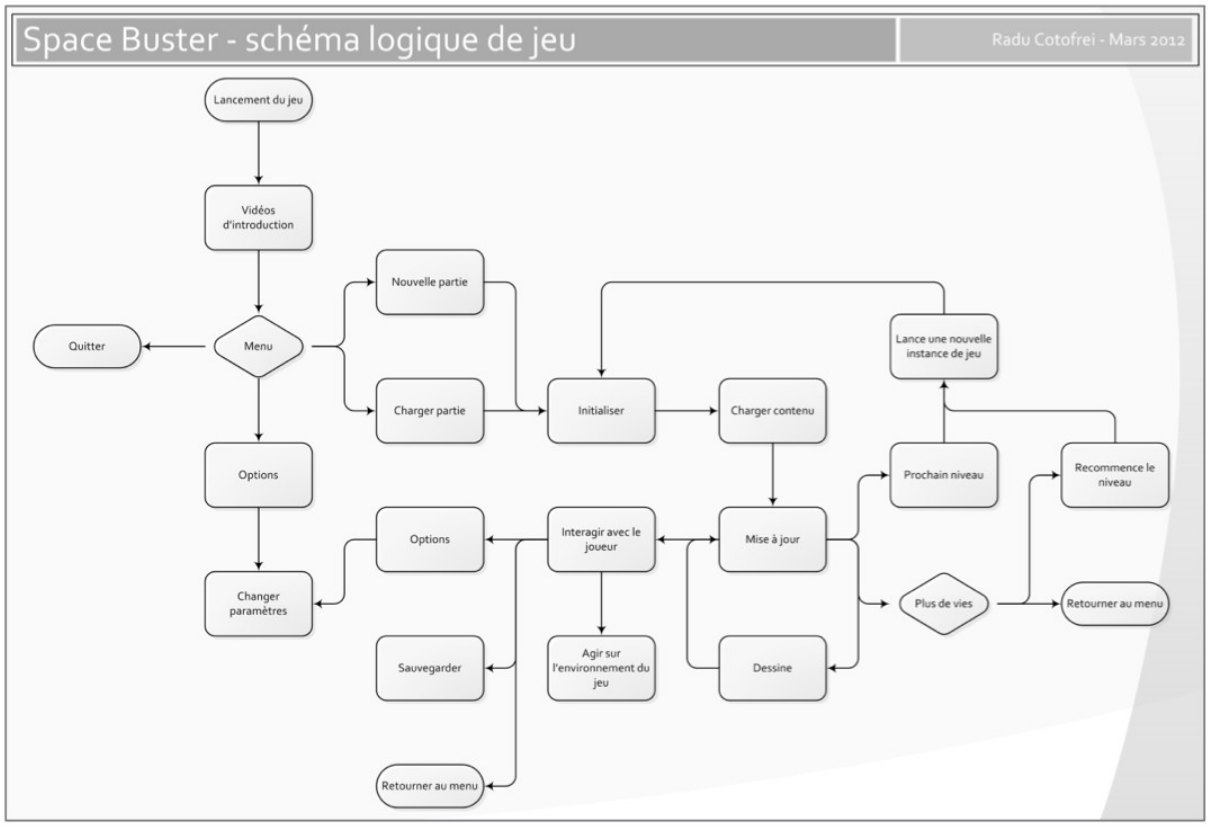

Fig 1. Schéma logique du jeu

Début du codage

Présentation d'un modèle de base

Le code de l'encadré 1 est généré au lancement d'un projet XNA avec Visual Studio.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
[...]
namespace EmptyGame {
  public class Game1 : Microsoft.Xna.Framework.Game {
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    public Game1() {
      graphics = new GraphicsDeviceManager(this);
      Content.RootDirectory = "Content";
    }
    protected override void Initialize() {
      base.Initialize();
    }
    protected override void LoadContent() {
      spriteBatch = new SpriteBatch(GraphicsDevice);
    }
    protected override void UnloadContent() {
    }
    protected override void Update(GameTime gameTime) {
      // Allows the game to exit
      if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();
      base.Update(gameTime);
    }
    protected override void Draw(GameTime gameTime) {
      GraphicsDevice.Clear(Color.CornflowerBlue);
      base.Draw(gameTime);
    }
  }
}
```

Encadré 1. Modèle de base

Expliquons un peu la structure de ce fichier.

Les différentes méthodes de la classe *Game1*

La méthode Initialize

C'est dans cette partie les variables seront initialisées (recevront des valeurs). Cette opération pourrait le faire lors de la déclaration, mais c'est plus lisible de cette manière. De plus, si par la suite on doit réinitialiser les variables de la classe *Game1*, il suffit d'appeler la méthode *Initialize*.

La méthode LoadContent

LoadContent permet de charger des ressources spécifiques au jeu, contenues dans des fichiers situées sur le disque dur, ou dans la mémoire vive. Etant donné que la dimension de la mémoire est limitée, il est judicieux de ne charger que ce qui est absolument nécessaire, puis de libérer de l'espace au moment où ces ressources ne sont plus utilisées. Il existe une méthode *UnloadContent* qui permet de libérer des ressources, en supprimant les contenus qui ne sont plus nécessaires.

La méthode Update

Les méthodes *Update* et *Draw* sont les méthodes les plus importantes pour notre application car elles sont au cœur même du jeu. En effet, un jeu vidéo n'est qu'une succession de tâches répétées, qui interagissent avec le joueur grâce à l'écran et aux périphériques. *Update* est appelé soixante fois par seconde et sert à mettre à jour nos variables, appeler des fonctions, etc... Elle a comme paramètre la variable *GameTime*, qui contient le temps de jeu global. Elle nous sert à contrôler le temps pour pouvoir gérer la logique et les actions du jeu. Par exemple, si l'on veut que le joueur qui tire avec le laser exécute cette action avec un retard de 0.5 secondes, on met une condition en utilisant la variable *gameTime*, à laquelle on aurait préalablement ajouté 0.5 secondes.

La méthode Draw

La méthode *Draw* permet de transcrire de manière graphique les éléments de notre programme. Comme pour la méthode *Update, Draw* est aussi appelée soixante fois par seconde et utilise aussi la variable *gameTime*.

Il est possible de changer la fréquence à laquelle *Update* et *Draw* sont appelées. De plus, cette fréquence peut être différente pour les deux méthodes. Pour information, cette fréquence, dans le langage du jeu vidéo, est appelée FPS (Frame Per Second) ou IPS (Images Par Secondes).

Créations des classes nécessaires

L'utilité des classes

Comme C# permet la programmation orientée objet, le plus simple est de créer des classes pour chaque objet que nous avons besoin, puis de les associer à notre code principal. Évidemment, on pourra se demander quelle est l'utilité de créer une classe pour la balle ou pour le vaisseau, alors qu'il n'y en a qu'un exemplaire ? En fait, la création de classes permet également d'alléger le code affiché à l'écran. Si par exemple on aimerait mettre à jour la balle en la déplaçant de 10 pixels vers la droite et de 5 pixels vers le bas, au lieu d'écrire :

```
Int balle_x = 0;
Int balle_y = 0;
balle_x += 10;
balle_y += 5;
```

il suffira d'écrire :

```
Ball balle = new Ball();
balle.Update(10,5);
```

L'objet *Ball* contient une méthode appelée *update* qui est définie une fois. Le code nécessaire au déplacement de la balle est donc contenu dans l'objet *Ball* et n'apparaît pas dans le code principal, ce qui l'allège.

Les principales classes

En tout, le jeu compte 36 classes. Par gain de temps, je vais commenter seulement les plus importantes : *Ball, Player* et *Target.*

La classe *Ball*

Caractéristiques

La balle du jeu est une animation d'une balle, qui se déplace dans la zone de jeu et rebondit contre les bords ou contre les obstacles. La classe *Ball* possède plusieurs propriétés dont : une animation, une position, une direction (vecteur normé), une vitesse, un rectangle délimitant son aire d'action, etc.

Le déplacement de la balle est défini par des principes physiques. A un moment donné, la balle est caractérisée par une position, un vecteur normé qui indique le sens et la direction, et un scalaire qui donne la vitesse. La nouvelle position n'est autre que l'ancienne position à laquelle on additionne le vecteur normé multiplié par la vitesse (voir figure 2).

$$\begin{pmatrix} X' \\ Y' \end{pmatrix} = \begin{pmatrix} X \\ Y \end{pmatrix} + \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \cdot k$$

$$avec \begin{pmatrix} X' \\ Y' \end{pmatrix} = nouvelle\ position, \begin{pmatrix} X \\ Y \end{pmatrix} = ancienne\ position, \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = vecteur\ normé, k = vitesse$$

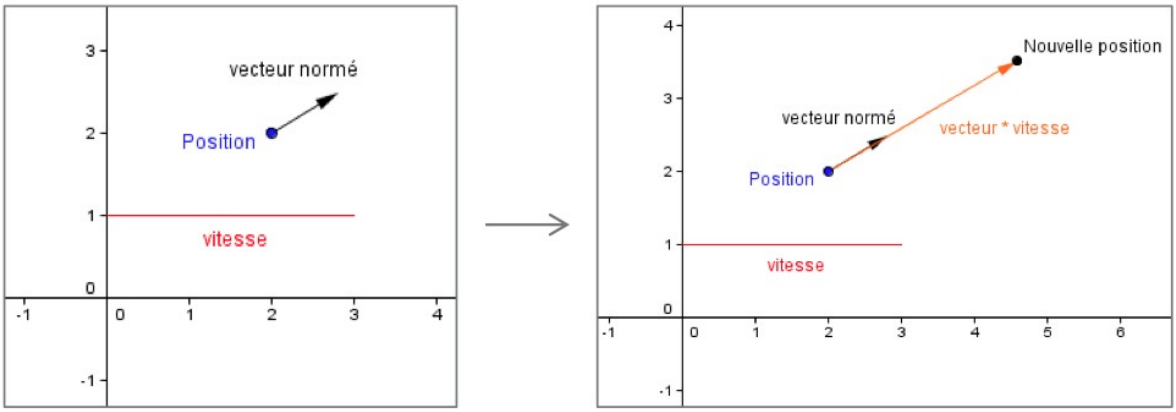

Fig 2. Le déplacement de la balle

Initialisation de la balle

Lorsque le joueur lance la balle, celle-ci doit partir avec un angle aléatoire, compris entre 30° et 60° ou 120° et 150° (choix arbitraire) (figure 3).

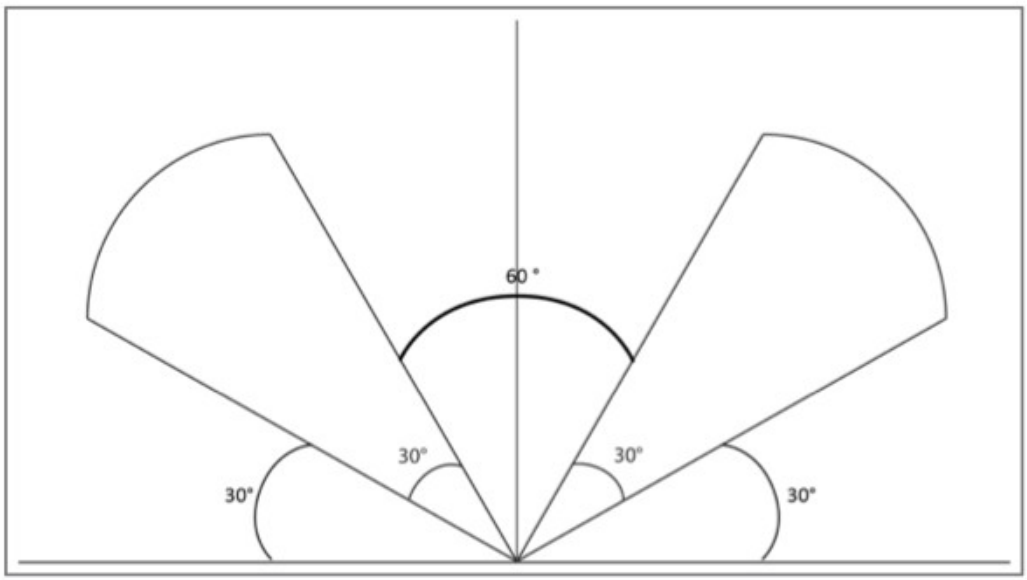

Fig 3. L'angle de la balle

La classe *PLAYER*

Caractéristiques

La classe *Player* représente le vaisseau contrôlé par le joueur. Il peut se déplacer horizontalement, dans l'espace de jeu délimité, mais également verticalement. Le vaisseau est là pour empêcher la balle de tomber dans la partie inférieure de la zone de jeu, sous peine de perdre une vie. Il doit de plus éviter ou détruire les cibles qui lui foncent dessus. Le vaisseau possède un laser et un bouclier, nécessitants de l'énergie pour fonctionner.

Le vaisseau est constitué de deux entités différentes. L'engin spatial a surtout un rôle décoratif[1], et la barre, qui est la pièce maîtresse. C'est sur la barre que va rebondir la balle et c'est également elle qui va activer les bonus.

Parmi les propriétés principales de cette classe, on peut citer : textures de la barre et du vaisseau, rectangles délimitant leur zone d'action, paramètres de jeu comme la vie, le score, etc..., les positions de la barre et du vaisseau, la vitesse avec laquelle le vaisseau se déplace, etc.

Contrôle du vaisseau par le joueur

Au clavier

Le vaisseau est contrôlé par le joueur qui décide de le faire avancer ou de tirer avec le laser. C# intègre une classe permettant de faire le lien entre le clavier de l'utilisateur et le programme. Cette classe s'appelle *Keyboard* et elle contient une méthode qui permet de connaître quel état possède chaque touche du clavier, dénommée *GetState*. Cette méthode appelle à son tour d'autres méthodes plus ciblées, comme *IsKeyDown*(Key) qui prend comme paramètre d'entrée une touche spécifique de notre choix du clavier. Il existe un tableau regroupant toutes les touches du clavier dénommé *Keys*. À partir de là, il devient simple de savoir ce que fait l'utilisateur et de réagir en conséquence. L'encadré 2 fournit un petit exemple.

```
//permet d'assigner à une variable l'état actuel du clavier
KeybordState currentKeyboardState = Keyboard.GetState();

//On vérifie si la touche "Enter" est appuyée
if(currentKeyboardState.IsKeyDown(Keys.Enter))
  Console.WriteLine("La touche enter est appuyée");
else
  Console.WriteLine("La touche enter est lâchée");
```

Encadré 2. Contrôle du clavier

Il faut tenir compte que le cycle de rafraîchissement du est exécuté 60 fois par secondes, ce qui peut entraîner des problèmes au niveau du *gameplay*. Il serait fâcheux de tirer 20 missiles laser, alors que nous n'avons appuyé qu'une seule fois sur la touche mais durant un laps de temps suffisant pour en tirer plusieurs. Pour régler ce problème, il suffit d'ajouter un *timer* qui contrôle qu'une touche n'est pas appuyée plusieurs fois de suite dans un intervalle donné. Pour cela, il suffit de modifier le code comme indiqué dans l'encadré 3.

```
/permet d'assigner à une variable l'état actuel du clavier
KeybordState currentKeyboardState = Keyboard.GetState();
//Initialisation du Timer
TimeSpan MomentPreviousPressKey;

//On vérifie si la touche "Enter" est appuyée
if(currentKeyboardState.IsKeyDown(Keys.Enter) &&
   gameTime.TotalGameTime - MomentPreviousPressKey > TimeSpan.FromSeconds(0.5)) {
  MomentPreviousPressKey = gameTime.TotalGameTime;
  Console.WriteLine("La touche Enter est appuyée");
} else {
 Console.WriteLine("La touche enter est lâchée");
}
```

Encadré 3. Contrôle du clavier amélioré

La variable *MomentPreviousPressKey* enregistre le moment lorsque l'utilisateur appuie sur une touche donnée (ici Enter). Ensuite, il suffit de vérifier si l'intervalle de temps entre ce moment (exprimé par

[1] À part s'il touche un objet, ce qui provoque la perte d'une vie.

l'attribut *TotalGameTime* de l'objet *gameTime*) et celui correspondant à la prochaine fois que la touche sera appuyée est supérieur à un intervalle choisi arbitrairement (ici 0.5 secondes). S'il est inférieur, on ignore la commande et on passe à la suite. Au contraire, s'il est supérieur, on lance la procédure correspondante et on réattribue une nouvelle valeur à *MomentPreviousPressKey* qui correspond au moment où la touche est appuyée.

Avec la manette Xbox 360

XNA reconnaît nativement la manette Xbox 360 et propose une classe déjà définie qui permet de l'utiliser de façon simple. La syntaxe étant quasiment identique que pour le clavier, on ne va pas entrer dans les détails. La classe *KeyboardState* est remplacée par *GamePadState,* qui demande un paramètre : le joueur. En effet, il est possible de gérer jusqu'à 4 joueurs, du coup il faut spécifier quel joueur contrôle quel vaisseau (malgré le fait que Space Buster ne gère pas le scénario multi-joueurs).

La classe *TARGET*

Caractéristiques

La classe *Target* (cible en français) représente un modèle de cible qui peut avoir différentes formes, différentes fonctions et différents attributs. Cette classe peut très bien simuler un astéroïde, une mine qui explose ou un ennemi qui nous tire dessus.

Cette classe possède plusieurs propriétés dont : une texture, une position, une vitesse, une variable qui définit le type de la cible, une variable fonction qui définit ce que la cible fera (par exemple, exploser au contact du vaisseau), une variable bonus qui définit le bonus de la cible, une variable santé, quatre rectangles (haut, bas, gauche, droite) gérant les collisions, etc.

Passage de données sous format XML

Les données des cibles sont stockées grâce au Game Editor (voir plus bas) dans un fichier XML [1]. Ensuite, lorsque le niveau charge, le jeu va créer un certain nombre de cibles grâce à ces données. L'encadré 4 montre un exemple de fichier XML décrivant deux cibles.

```
<Targets>
  <Item>
    <position>583 -7680</position>
    <type>mine</type>
    <texture>mine_texture</texture>
    <function>Explose</function>
    <bonus>SlowerTargets</bonus>
    <health>100</health>
  </Item>
  <Item>
    <position>265 -7520</position>
    <type>asteroide1</type>
    <texture>asteroide1_texture</texture>
    <function>None</function>
    <bonus>None</bonus>
    <health>100</health>
  </Item>
</Targets>
```

Encadré 4. Codage des données des cibles

La balise *Targets* contient les paramètres de toutes les cibles du niveau. Chaque balise *Item* représente une cible. Lorsque le jeu charge le niveau, il ajoute autant de cibles que contient la balise *Targets,* avec les caractéristiques propres à chaque cible.

Gestion des collisions

La gestion des collisions dans un casse-briques est l'un des aspects les plus importants du jeu. Malgré les aides que XNA peut nous fournir pour nous permettre à les gérer, il est nécessaire d'avoir quelques notions de physique et de représentation des objets dans l'espace. Comme le peintre, le développeur de jeu doit bien observer et comprendre la réalité qui l'entoure pour pouvoir la reproduire aussi fidèlement que possible dans son jeu. On pourrait penser que c'est chose aisée de gérer le comportement physique des objets, mais ce n'est point le cas. Heureusement, XNA intègre dans ses bibliothèques une classe *Rectangle* qui va nous simplifier la tâche. Il suffit d'attribuer un rectangle à

[1] XML, Extensible Markup Language, est un langage permettant de structurer des documents à l'aide de balises.

chaque objet et de détecter s'il entre en contact avec le rectangle d'un autre objet, grâce à la méthode *Intersects* de la classe *Rectangle*. Pour la balle, deux méthodes sont utilisées : l'utilisation d'un seul grand rectangle, pour savoir si la balle touche un autre objet, et la gestion précise de l'endroit où la balle a touché l'objet grâce à plusieurs petits rectangles qui entourent la balle.

On distingue plusieurs types de collisions, notamment entre la balle et la zone de jeu, la balle et la barre, la balle et les cibles et la barre et les bonus.

Collisions entre la balle et la zone de jeu

La balle, ainsi que les quatre bords de la zone de jeu, sont représentés par des rectangles, comme illustré dans la figure 4.

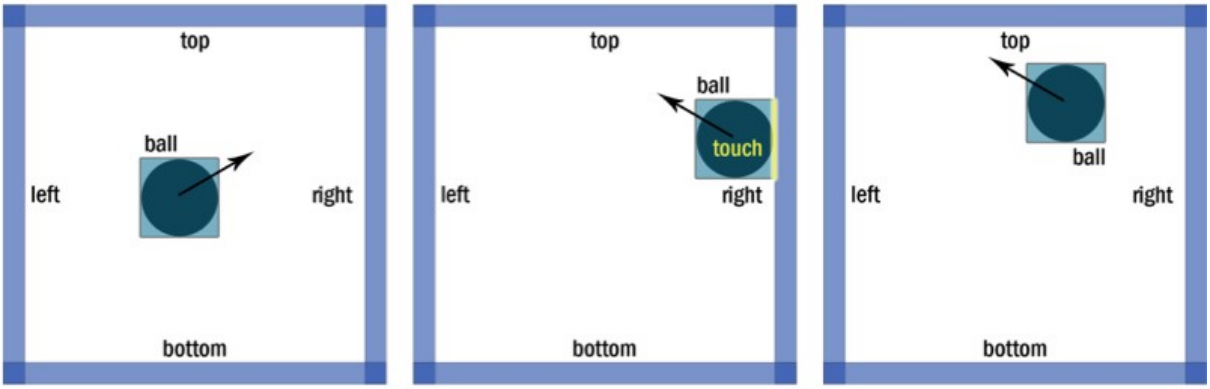

Fig 4. Les rectangles gérer le mouvement de la balle dans l'espace de jeu

Si le rectangle de la balle entre en contact avec l'un des quatre rectangles de la zone de jeu, alors la composante verticale (bord du haut) ou horizontale (bords droit et gauche) du vecteur vitesse de la balle sera égale à son opposé. S'il entre en contact avec le rectangle du bas, alors la balle sera réinitialisée et le joueur perdra une vie.

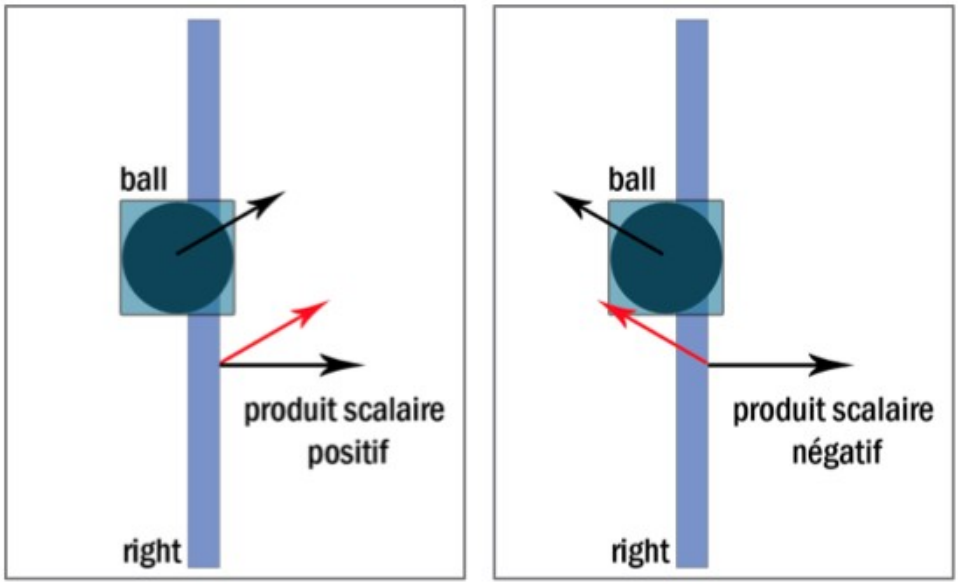

Fig 5. Contrôle de la direction de la balle

Cependant, malgré le fait que la détection se fasse à chaque étape, il arrive que la balle reste « coincée » dans un bord. En effet, si le rectangle de la balle, à cause d'un problème d'arrondi, dépasse une certaine limite, alors il ne peut plus échapper du bord car le pas qu'il fait dans le sens contraire est trop petit. Cela signifie qu'il change continuellement sa composante vitesse. Pour parer à ce problème, on va utiliser une notion mathématique : le produit scalaire. Pour chaque bord, on définit un vecteur qui est orienté vers l'extérieur. Lorsque la balle entre en collision avec un bord, il suffit de calculer le produit scalaire entre le vecteur normal du bord et le vecteur vitesse et d'observer son signe. Si le signe est positif, cela indique que la balle se dirige vers l'extérieur, et donc sa direction doit être

changée. Si le signe est négatif la balle se dirige vers l'intérieur de la zone de jeux, donc rien ne doit être entrepris. La figure 5 illustre cette situation.

Pour finir, l'encadré 4 propose un fragment du code concernant la collision avec le bord de droite.

```
if (Ball.Rectangle.Intersects(GameZone.BorderRight)) {
  //on initialise le vecteur normal au bord droit
  Vector2 right = new Vector2(1, 0);

  if (Vector2.Dot(Ball.DirectionVector, right) >= 0) {
    //on oppose le sens de la composante horizontale
    Ball.DirectionVector.X *= -1;

    //on produit le son généré lorsque la balle touche le bord
    Sound CollisionSound = new Sound();
    CollisionSound.Initialize("Collision_3", "Effects", false);
    SoundList.Add(CollisionSound);
  }
}
```

Encadré 4. Code concernant la collision

Collisions entre la balle et la barre

Lorsque la balle entre en contact avec la barre, il faut que l'angle avec lequel elle repart varie en fonction de l'endroit où elle a touché la barre. Plus la balle est près du centre de la barre, plus l'angle de départ sera proche de 90°. Inversement, plus elle est proche des bords, plus l'angle approchera 30° ou 150° (cf. classe *Ball)*. Pour détecter si la balle touche la barre, il suffit de réutiliser la méthode *Intersects* citée plus haut. Une fois que la balle est en contact avec la barre, il faut calculer l'angle de départ grâce aux rapports (encadré 5).

```
if (Ball.Rectangle.Intersects(Player.BarreRectangle)) //Si la balle touche la barre {
  float rapport; //On déclare la variable rapport
  rapport = (Ball.Center.X - (float)Player.BarrePositionCorner.X) / (float)Player.BarreWidth;
  rapport = 1 - rapport; //Pour que l'angle soit celui qu'on veut, il faut prendre l'opposé
du rapport
  rapport = MathHelper.Clamp(rapport, 0f, 1f); //Permet de toujours avoir un rapport entre 0
et 1

  //Avant que la balle ne soit lancée, elle touche la barre. Il faut vérifier qu'elle bouge.
  if (Ball.move) {
    degree = (int)(120 * rapport + 30); //On dit que l'angle couvre de 30° à 150°
    Ball.ChangeDirection(degree); //On donne un nouvel angle à la balle
  }
}
```

Encadré 5. Calcul de l'angle de départ

La fonction *Clamp* de la classe *MathHelper* est très pratique car elle permet de renvoyer la valeur d'un nombre dans un intervalle donné.

Collisions entre la balle et les cibles

Dans le cas de la collision de la balle avec les cibles, il ne faut plus seulement savoir s'il y a contact, mais également connaître l'endroit précis où la collision a eu lieu. Pour cela, il faut utiliser plusieurs petits rectangles formant le contour de la balle et vérifier chacun (dans la figure 6 seuls 8 rectangles parmi les 18 au total sont représentés).

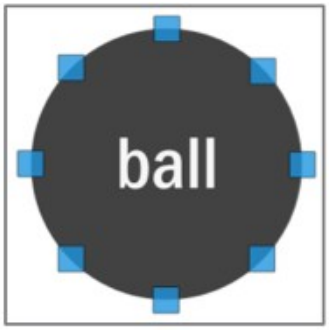

Fig 6. Les rectangles détecteurs de collisions

Si la collision a lieu avec les 3 rectangles du haut ou les 3 rectangles du bas, alors ce sera la composante verticale du vecteur vitesse de la balle qui sera opposé. De façon similaire, la composante horizontale du vecteur vitesse va changer de direction si la collision a lieu avec les 3 rectangles à gauche ou à droite.

Une autre méthode consiste à détecter quel partie de la cible est touchée : si c'est la partie supérieure ou inférieure, alors la balle changera de direction verticale. Même scénario avec les parties latérales, sauf que la balle change de direction horizontale.

Dès que la balle entre en contact avec une cible, il faut créer et faire apparaître le bonus (s'il y en a un), augmenter le score et détruire la cible en la supprimant de la liste des cibles actives (si sa barre de vie est descendue à zéro).

Collisions entre la barre et les bonus

La méthode de détection est identique à ce qui a déjà été fait jusqu'à maintenant : la barre et les bonus possèdent des rectangles et on cherche à déterminer s'ils entrent en contact ou pas. Ce qui change, c'est que chaque bonus possède un attribut qui détermine une action sur le joueur. Par exemple, si le joueur touche le bonus « plus de vie », il faut qu'il ait une vie de plus qui s'ajoute à ses vies. Dès qu'on détecte une collision, on passe l'attribut du bonus entré en collision. Ensuite, on cherche quel attribut possède le bonus et on lance une action.

Game Editor

Généralement, les jeux ne sont plus codés comme à l'ancienne, « à la main ». En fait, de nos jours, les développeurs professionnels et amateurs utilisent des moteurs graphiques qui leur permettent de créer des jeux « à travers un logiciel dédié ». Si je reprends la définition de Wikipédia :

Un moteur de jeu est un ensemble de composants logiciels qui effectuent des calculs de géométrie et de physique utilisés dans les jeux vidéo. L'ensemble forme un simulateur en temps réel souple qui reproduit les caractéristiques des mondes imaginaires dans lesquels se déroulent les jeux. Le but visé par un moteur de jeu est de permettre à une équipe de développement de se concentrer sur le contenu et le déroulement du jeu plutôt que la résolution de problèmes informatiques. (Wikipédia, Moteur de jeu, 2012).

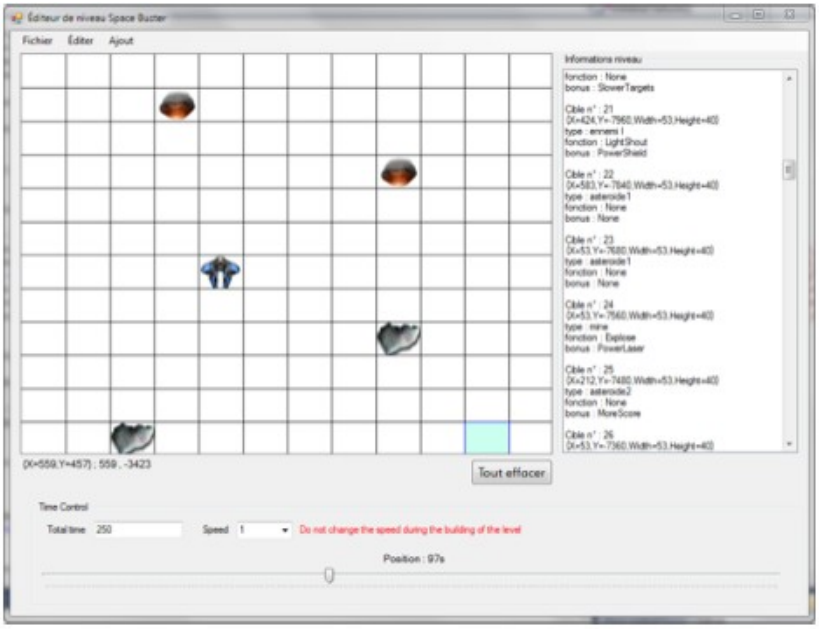

Fig 7. Edition des cibles pour un niveau de jeu

Parmi les moteurs de jeux connus et célèbres il y a le *CryEngine 3* de Crytek[1], *l'Unreal Engine 3* d'Epic Games[2] ou encore le *Frostbite Engine 2* de chez DICE[3]. Il existe un nombre impressionnant de

[1] http://mycryengine.com (consulté : décembre 2012)

[2] http://www.unrealengine.com/ (consulté : décembre 2012)

moteurs de jeux, qu'ils soient gratuits ou payant. Chaque éditeur a son propre moteur graphique développé en interne. L'objectif principal est de bien le maîtriser en d'en tirer le meilleur parti.

Game Editor pour Space Buster

Pour mon jeu, il aurait été trop difficile de créer un vrai moteur de jeu. Quoi qu'il en soit, j'ai toutefois développé un petit logiciel qui permet de créer des niveaux de manière simple. En effet, il suffit de choisir quelques paramètres, comme le fond ou la musique, puis de placer les cibles (figure 7). Il est possible de supprimer des cibles ou de les modifier. La barre du temps permet d'avancer ou de reculer le temps relatif du jeu et de savoir exactement à quel moment une cible apparaîtra à l'écran. Lorsque la création d'un niveau est terminé, il ne reste plus qu'à le sauvegarder en format XML. Le jeu intégrera automatiquement le niveau au prochain démarrage.

Paramètres de niveau et des cibles

Pour chaque niveau, on peut choisir son numéro, la musique qui sera jouée pendant la partie (et si en boucle) et la vidéo qui tournera en arrière-plan.

Pour les cibles, il y a d'abord le type de cible (Astéroïde 1 ou 2, Mine, Ennemi 1, 2 ou 3), puis la l'action à déclencher (si c'est une mine, elle doit exploser si on la touche), le bonus (plus de vie, barre plus grande, etc.) et pour finir la résistance de la cible, sa capacité à encaisser les chocs de la balle ou du laser (figure 8).

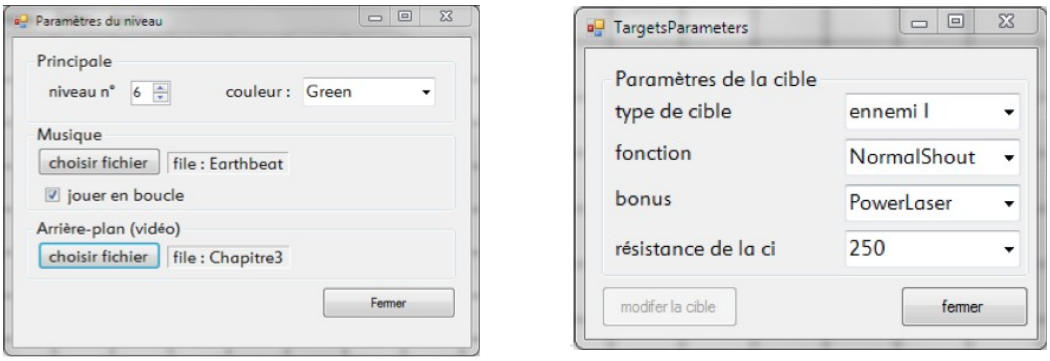

Fig 8. Paramètres de niveau et des cibles

Le mode aléatoire

On se rend vite compte que placer chaque cible une à une devient compliqué et que ça prend beaucoup de temps. Pour remédier à ça, j'ai créé un mode aléatoire qui place automatiquement des cibles au hasard. Le créateur n'a qu'à choisir les paramètres souhaités pour le niveau de jeu et les enregistrer. Les paramètres à disposition sont : « densité » de cibles, le pourcentage de chaque type de cible, les bonus présents et le pourcentage des cibles avec un bonus (figure 9).

Pour bien comprendre le paramètre « densité », il faut se représenter le niveau non pas comme une matrice de rectangles, mais comme un vecteur ligne obtenu par linéarisation de la matrice. Quand la fonction place une cible, elle en met une au hasard entre *n* et *m* cases plus loin de la position courante (où *n* et *m* sont choisis par l'utilisateur) (figure 10).

Conclusion

En définitive, le but est atteint et le jeu fonctionne. Mais un jeu n'est jamais fini. Je continuerai à y travailler et je proposerai des mises à jour téléchargeables depuis internet.

L'expérience acquise me sera également d'une grande utilité lors de la réalisation d'autres jeux. A ce propos, je note que dans ce projet le plus difficile a été la gestion des collisions. Adapter un phénomène physique réel au monde virtuel n'est pas une mince affaire et de ce point de vue le projet serait à améliorer.

Cette relative « faiblesse » du projet est le résultat d'un choix : créer mon propre « moteur » plutôt qu'implémenter un moteur physique gratuit disponible sur internet.

[3] http://www.dice.se/ (consulté : décembre 2012)

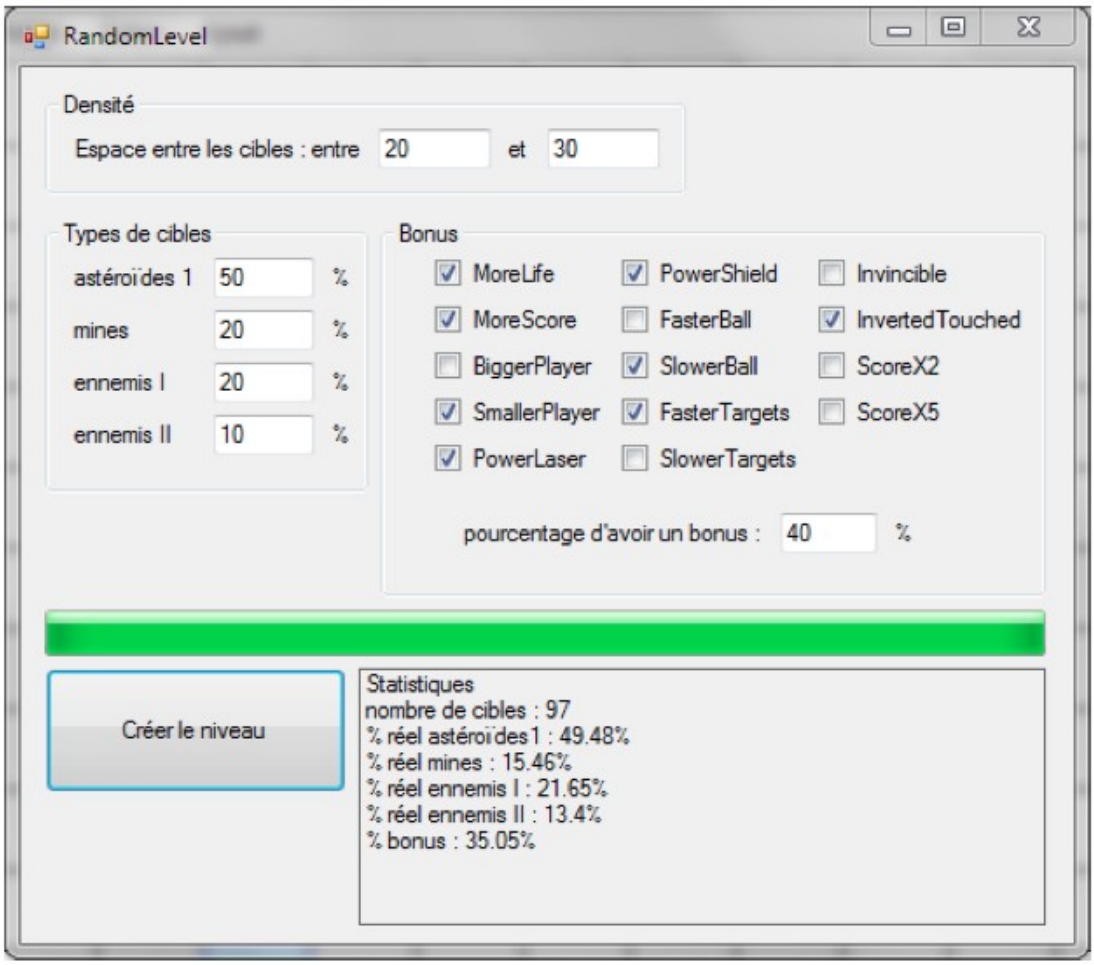

Fig 9. Choix des paramètres pour un niveau de jeu

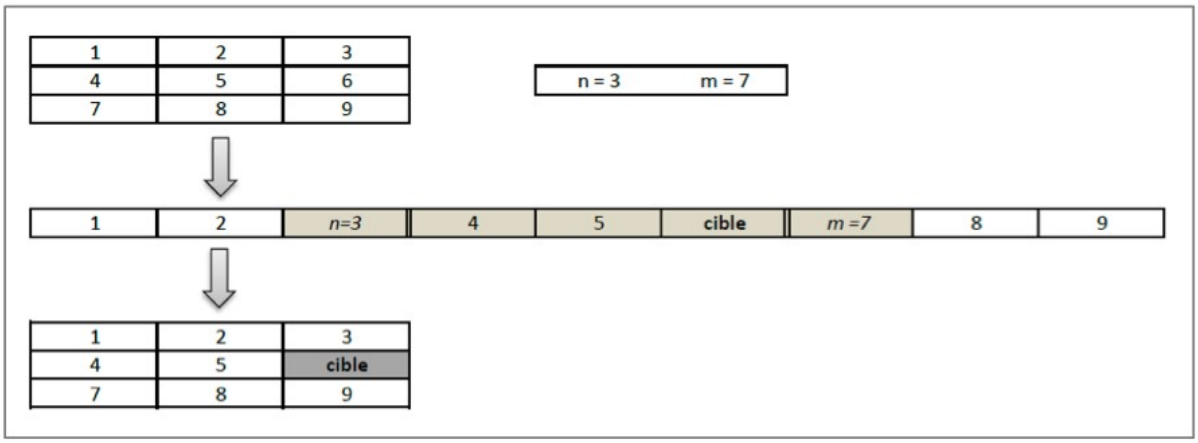

Fig 10. Placement d'une cible

Fig 11. Résultat final

Bibliographie

Game State Management. (2011, 2 Mai). http://create.msdn.com/en-US/education/catalog/sample/game_state_management (consulté : novembre 2012).

Site officiel de Space Buster. (2012). http://www.rightlightmusic.com/SpaceBuster/ (consulté : novembre 2012).

App Hub. (s.d.). Récupéré sur MSDN Create: http://create.msdn.com/en-US/ *Apprenez à programmer en C# sur .NET.* (s.d.). http://www.siteduzero.com/tutoriel-3-344102-apprenez-a-programmer-en-c-sur-net.html (consulté : novembre 2012).

Crytek. (s.d.). *Cry Engine 3*. http://mycryengine.com/ (consulté : novembre 2012).

DICE. (s.d.). *Frostbite Engine 2*. http://www.dice.se/ (consulté : novembre 2012).

Games, E. (s.d.). *Unreal Engine 3*. http://www.unrealengine.com/ (consulté : novembre 2012).

Microsoft Visual Studio C# Express. (s.d.). http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express (consulté : novembre 2012).

Wikipédia. (2009, Juillet 30). *Visual C Sharp.* http://fr.wikipedia.org/wiki/Visual_C_Sharp (consulté : 24 Mars 2012)

Wikipédia. (2012, Mars 21). *C Sharp.* http://fr.wikipedia.org/wiki/C_sharp (consulté : 24 Mars 2012)

Wikipédia. (2012, Janvier 26). *Microsoft XNA.* http://fr.wikipedia.org/wiki/Microsoft_XNA (consulté : 24 Mars 24 2012)

Wikipédia. (2012, Janvier 12). *Moteur de jeu.* http://fr.wikipedia.org/wiki/Moteur_de_jeu (consulté : 24 Mars 2012)