École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
3e année
2011 - 2012

Rapport de projet d'Algorithmique / Langage C

Développement d'un jeu Bloxorz sur Xbox360

Encadrants
Carl ESSWEIN
carl.esswein@univ-tours.fr
Jean-Louis BOUQUARD
jean-louis.bouquard@univ-tours.fr

étudiants
Benoit BELZ
benoit.belz@etu.univ-tours.fr
Fabien FARIN
fabien.farin@gmail.com

Université François-Rabelais, Tours

DI3 2011 - 2012

Version du 14 juin 2012

Table des matières

Table des figures

Liste des tableaux

Introduction

Dans le cadre du projet d'algorithmique et langage C nous avons mis en oeuvre un portage du jeu Bloxorz sur la console de salon la Xbox 360 de la société Microsoft. Ce projet permet de mettre en oeuvre les connaissances acquises lors du cours d'algorithmique et lors du cours de langage C dans un projet pratique avec une problèmatique à résoudre. Ce rapport est là pour montrer la démarche de création du jeu que nous avons utilisée afin de mener à bien ce pojet. Nous verrons dans un premier temps la démarche algorithmique puis l'aspect implémentation du projet.

Présentation du sujet

Ce projet à pour but de créer une version Xbox 360 du jeu Bloxorz, jeu de réflexion en flash développé par Damien Clarke en 2007 pour le compte de la société DX Interactive. Le but du jeu est de déplacement un bloc, de faọn à rejoindre un trou qui déclenche la fin du niveau. La faọn dont le joueur déplace son personnage en analysant le terrain permet de trouver la solution.
La Xbox 360 est la seconde génération de console de salon développé par Microsoft. Sortie en 2006, elle posséde des capacités de traitement impressionnants pour l'époque. Cette console devient de plus en plus populaire, surtout avec la sortie du Kinect. Si cette console est intéressante pour dévelloper, c'est parce que Microsoft permet aux devellopeurs d'avoir des outils permettant de programmer la console. Ces outils sont accessibles et bénéficient des technologies de Microsoft. C'est aussi l'occasion pour nous d'apprendre la programmation d'un jeu et d'avoir le résultat sur une console de salon grand public.

Cahier des charges

3.1 Le cahier des charges

Notre cahier des charge était le suivant :
- Nous devions réaliser une copie du jeu flash Bloxorz pour XBOX 360. Elle doit garder les fonctionnalités du jeu de départ.
- Notre jeu devait permettre la possibilité d'ajouter un éditeur de Niveaux, même si nous ne le réalisions pas, la structure devait permettre une implémentation facile de celui-ci.
- Le jeu devait être jouable avec une manette de XBOX.

3.2 Les contraintes

- Le jeu sera en 3D.
- Le développement de jeu sur XBOX 360 ce fait en utilisant le langage C# et de le framework XNA de Microsoft pour VisualStudio.

3.3 Fonctionnalités de départ

Voici les fonctionnalités de départ :
- Le jeu est composè de plusieurs niveaux
- Le joueur est un bloc de 2 sur 1 sachant que 1 représente la longueur d'un côté d'une dalle dans un niveau
- Le joueur peut activer des actions, soit pour faire apparaître ou disparaître des dalles soit pour séparer le joueur en deux
- Il existe des dalles ne supportant le joueur que si celui-ci est allongé
- Si le joueur se déplace sur un emplacement en dehors des dalle, c'est une fin de jeu (game over)
- La fin d'un niveau est délimité par l'accés à un trou dans le niveau où le joueur doit être dessus debout

Algorithmes du jeu

4.1 Démarche principale

Afin de définir les algorithmes de jeu il d'abord réfléchir sur la décomposition du jeu et de son déroulement.
Le jeu en lui-méme est un enchaénement de niveaux. Si le joueur arrive é terminer un niveau il passe au niveau suivant, sinon il recommence celui-ci. A la fin, lorsque le dernier niveau du jeu est terminé, c'est la fin du jeu.
Le jeu en lui-méme comporte un seul joueur, un monolite rectangulaire dont la longueur est égale é deux fois la largueur. Une des spécificités est qu'il puisse ce séparer en deux, si l'action associé est appliquée dans le cadre d'une action disponible dans le niveau.
Un autre point que nous avons pris en compte lors de la conception des algorithmes pour l'implémentation, est que nous travaillons avec un langage objet. Chaque elément du jeu, y compris le jeu en lui-méme seront des instances de classes ou interfaces définies. Donc les algorithmes de gestions deviendront des méthodes propres aux classes des objects auquels ils sont appliqués. Pour la conception algorithmique, chaque élément sera donc défini comme un module.

L'approche algorithmique s'effectuera donc sur l'exécution d'un niveau avec un joueur en général, avec les spécificitées qu'il peut comporter. Nous allons donc commencer par voir l'algorithme de déroulement d'un niveau pour passer aux algorithmes de gestion du joueur et des actions/evénements qui seront présents dans un niveau.

4.2 Déroulement d'un jeu (niveau)

C'est la boucle principale de gestion d'un jeu lorqu'un niveau est lancé. Il utilise une variable Jeu actif et Etatjeu qui indique dans quel état ce trouve celui-ci pour effectuer les actions associés aux differentes entitées du jeu.

Algorithme 1 Boucle de jeu

Précondition: entrées : $numéroduniveau, nombredetentative, nombredemouvementdéjàeffectué$
Postcondition: Le joueur à fini le niveau ou doit le recommencer

```
    /* Initialisations */
    niveauEnCours ← numéroduniveau
2:  nombreTentative ← nombredetentative
    nombredemouvement ← nombredemouvementdéjàeffectué
4:  nombreDeMouvementCourant ← nombredemouvementdéjàeffectué
    /* Chargement des médias externes */
    Cube ← Charger(ModeleCube3D)
6:  Sons ← Charger(Sons)
    /* Création du jeu */
    niveau ← CreerNiveau(niveauEnCours)
8:  joueur ← CreerJoueur(niveauEnCours ↑ .departjoueur)
    Etatdujeu ← initialisation
10: Evenementmanette ← Initialisation()
    ConfigurerMateriel()
12: InitialisationParametresCamera3D()
    /* Boucle de jeu */
    tantque JeuActif faire
14:     MiseAJourEvenementManette()
        switch (Etat du jeu)
16:     case Initialisation:
            initialiserJeu()
18:     case début niveau animation:
            AnimationDebutNiveau()
20:     case Animation:
            MiseAJourAminationEnCours()
22:     case Jeu:
            si Les evénements du niveau ont été vérifiés alors
24:             MiseAJourDuJoueur(EvenementManette)
                VerificationFinDeplacementJoueur()
26:         sinon
                MiseAJourDuNiveau(joueur)
28:             Lesevénementsduniveauontétévérifiés
            fin si
30:         RéinitialisationDesEvenementsManettes()
        case Game Over:
32:         Lancer le méme niveau avec les paramétres d'entrée de celui-ci
        case Fin du niveau:
34:         Lancement du niveau suivant avec le nombre de mouvement courant et de tenta-
            tive+1
        end switch
36: fin tantque
```

4.3 Le joueur

Le joueur est une entitée comportant des variables :
- Cube : tableau [1,2] de cube

- Etat : variable comportant deux états, un_morceau, deux_morceau
- cube_selectionné : variable comportant deux états, un ou deux
- mouvement_en_cours : booléen
- temps_passé : entier
- temps_mouvement : entier

Un cube est une structure comportant les variables suivantes :

- X : réel, coordonnée en X du cube
- Y : réel, coordonnée en X du cube
- Z : réel, coordonnée en X du cube
- rX : réel, rotation en X du cube
- rY : réel, rotation en Y du cube
- rZ : réel, rotation en Z du cube

4.3.1 Mise à jour du joueur

Prend en compte les evénements de la manette pour le répercuter sur le joueur si c'est possible.

Algorithme 2 Mise a jour du joueur

Précondition: entrées : les evénements de la manette

Postcondition: Les evénements ont été pris en compte si l'état du joueur le permet

```
1:
2: si le joueur est en deux morceaux et que l'action de changer le cube selectionné est demandé alors
3:         Onchangelaselectionducube
4:
5: fin si
6:
7: si Etat du jeu = jeu et !deplacement_en_cours alors
8:
9:         si Evenement deplacement haut alors
10:                DeplacementJoueurHaut()
11:
12:        sinon
13:
14:                si Evenement déplacement bas alors
15:                        DeplacementJoueurBas()
16:
17:                sinon
18:
19:                        si Evenement déplacement gauche alors
20:                                DeplacementJoueurGauche()
21:
22:                        sinon
23:
24:                                si Evenement déeplacement droite alors
25:                                        DeplacementJoueurDroite()
26:
27:                                fin si
28:
29:                        fin si
30:
31:                fin si
32:
33:        fin si
34:
35: sinon
36:
37:        si Un déplacement est en cours alors
38:                MiseAJourDuMouvement()
39:
40:        fin si
41:
42: fin si
```

4.3.2 Mise à jour du mouvement

Comporte tous les algorithmes concernant la mis à jour d'un mouvement sur le joueur déja initié.

Algorithme 3 MiseAJourDuMouvement

Précondition: entrées : *étatdujoueur*

Postcondition: la mise à jour du mouvement en cours

1:
2: **si** Etat = un_morceau **alors**
3: Mise à jour du joueur en un morceau()
4:
5: **sinon**
6: Mise à jour mouvement joueur en deux morceaux()
7:
8: **fin si**

Bien que peu utilise, cette fonction permet d'alléger l'analyse algorithmique et d'avoir une meilleure visibilité.

Algorithme 4 Mise à jour du joueur en un morceau ou deux morceaux

Précondition: entrées : variables du joueur et de gestion du mouvement courant

Postcondition: la mise à jour du mouvement en cours

1:
2: **pour** chaque cube impliqué dans le mouvement demandé **faire**
3:
4: **si** temps_passé < temps_mouvement **alors**
5: temps_passé++
6: Pour chaque rotation de l'objet définit lors de l'initialisation du mouvment on met à jour en fonction du temps passé
7:
8: **sinon**
9: On réinitialise le temps passé au mouvement à zéro
10: Pour chaque translation effective définies lors de l'initialisation du mouvement on met la valeur finale
11: On réinitialise les rotations et la position du point d'origine du modéle 3D
12:
13: **fin si**
14:
15: **fin pour**

4.3.3 Initialisation du mouvement

Deux méthodes sont utilisées pour l'initialisation d'un mouvement. La première est reservée à l'état joueur en deux cubes et la seconde au joueur en un seul morceau. Bien que ce soit deux fonctions diférentes, la structure algorithmique reste la même.

Algorithme 5 Initialisation du mouvement du joueur en un morceau ou deux morceaux

Précondition: entrées : variables du joueur et de la gestion de mouvement
Postcondition: L'initialisation du mouvement en cours

```
1:
2: pour chaque cube impliqué dans le mouvement demandé faire
3:
4:     pour chaque axe de déplacement faire
5:
6:         si un déplacement sur l'axe est définit alors
7:             On définit le mouvement pour l'axe
8:
9:         fin si
10:        On enregistre le mouvement dans une variable MouvementCourant
11:
12:    fin pour
13:
14: fin pour
```

4.3.4 Vérification fin du mouvement

L'algorithme suivant permet de détecter la fin d'un mouvement et d'effectuer les actions necéssaires car il faut pouvoir revérifier que le joueur est toujours en position de continuer le jeu.

Algorithme 6 Vérification de la fin du mouvement du joueur

Précondition: entrées : *variablesdujoueur*
Postcondition: Si une fin de mouvement est détectée, alors elle sera signalée et le nombre de mouvement sera augmenté de 1

```
1: /* Détection de la fin de mouvement */
2:
3: si variable actuelle de déplacement = faux et ancienne variable de déplacement = vraie alors
4:
5:     si Le joueur est en deux morceaux alors
6:         Reconciliation()
7:         Incrémentation du nombre de mouvement
8:         Evénements niveau vérifiées = faux
9:
10:    fin si
11:
12: fin si
```

Le prochain algorithme permet de réconcilier le joueur séparé en deux en un seul bloc si les conditions sont réunis.

Algorithme 7 Reconciliation

Précondition: entrées : *variablesdujoueur*

Postcondition: Si le joueur est séparé en deux cubes et que les cubes sont dans le prolongement direct l'un de l'autre, le joueur redevient un seul bloc

```
1:
2:  si Les deux cubes du joueur sont à la même coordonnée sur le plan X alors
3:
4:      si Les deux cube sont à une coordonnée +-1 sur le plan Z alors
5:          Le joueur redevient un bloc
6:
7:      fin si
8:
9:  fin si
10:
11: si Les deux cubes du joueur sont à la même coordonnée sur le plan Z alors
12:
13:     si Les deux cubes sont à une coordonnée +-1 sur le plan X alors
14:         Le joueur redevient un bloc
15:
16:     fin si
17:
18: fin si
```

4.3.5 Les actions de mouvement

On a vu dans l'algorithme de mise à jour du joueur que des fonctions DeplacementJoueurXXX peuvent être appellées. Ces fonctions ont toutes la même structure algorithmique, seul les tests changes. On va donc voir l'algorithme de la la fonction DeplacementJoueurHaut.

Algorithme 8 DeplacementJoueurHaut

Précondition: entrées : *variablesdujoueur*

Postcondition: Fait appel à l'initialisation du mouvement correspondant suivant les variables de position et d'état du joueur

```
1:
2: switch (etat du joueur)
3:
4: case Le joueur est en un morceau:
5:
6:     si Les deux cubes ont la même coordonnée sur l'axe X alors
7:
8:         si Les deux cubes ont la même coordonnée sur l'axe Z alors
9:
10:            si La coordonnée sur l'axe Y du premier cube est inférieure à la coordonnée
               sur l'axe Y du second cube alors
11:                Initialisation animation deux cubes(joueur debout premier cube en
                   bas,haut)
12:
13:            sinon
14:                Initialisation animation deux cubes(joueur debout deuxieme cube en
                   bas,haut)
15:
16:            fin si
17:
18:        sinon
19:
20:            si La coordonnée sur l'axe Z du premier cube est inférieure à la coordonnée
               sur l'axe Z du second cube alors
21:                Initialisation animation deux cubes(joueur allongé premier cube der-
                   nière,haut)
22:
23:            sinon
24:                Initialisation animation deux cubes(joueur allongé deuxiéme cube de-
                   rière,haut)
25:
26:            fin si
27:
28:        fin si
29:
30:    sinon
31:        Initialisation animation deux cubes(le joueur est allongé les deux cubes sont ali-
           gnés,haut)
32:
33:    fin si
34:
35:
36: case Le joueur est en deux morceau:
37:     Initialisation animation un cube (CubeSelectionne,haut) ;
38:
39:
40: end switch
```

4.4 Le niveau

C'est le module qui va gérer les Niveaux du jeu, ils contiennent les informations permettant de dessiner le Niveau à l'écran, ainsi que les Evenements (détaillés ailleurs).

Ce module possède la structure suivante :
Attributs :
Entier largeurCarte
Entier longueurCarte
Dalle listeDalles[,]
Entier nombreEvenements
Evenement listeEvenements[]
CoordoneesDepart
Méthodes :
GenerationDuNiveau()
DessinerNiveau()

GenerationDuNiveau()
C'est la méthode qui est appelée au moment du chargement du Niveau. Elle va initialiser les attributs du module.

Algorithme 9 Chargement Niveau

$largeurCarte \leftarrow ?$
$longueurCarte \leftarrow ?$

pour I de 1 ? largeurCarte **faire**

pour J de 1 ? longureurCarte **faire**
$listeDalles[I, J] \leftarrow VIDE$

fin pour

fin pour

pour chaque Dalle que l'on souhaite PLEINE [I,J] **faire**
$listeDalles[I, J].type \leftarrow PLEINE$

fin pour
$coordonneesDepart.x \leftarrow ?$
$coordonneesDepart.y \leftarrow ?$
$nombreEvenements \leftarrow ?$

pour I de 1 ? nombreEvenements **faire**
$listeEvenements[I].type \leftarrow IemeEvénement$

fin pour

MettreAJourNiveau()

C'est la méthode qui sera appelé par le module Jeu pour mettre Ã jour le Niveau (si il y besoin de le faire) après un déplacement du Joueur.
La séparation en module fait que cette fonction est très courte.

Algorithme 10 Mettre a Jour Niveau

```
pour TOUS les Evenements de Niveau.listeEvenements AS unEvent faire
    checkerEvenement(unEvent, Joueur)

fin pour
```

DessinerNiveau()
Cette méthode est appelée par la boucle principale du jeu. Elle sert à afficher la terrain du jeu à l'écran. On utilise la méthode de dessin du module Dalle.

Algorithme 11 Dessiner Niveau

```
1:
2: pour I de 1 ? largeurCarte faire
3:
4:     pour J de 1 ? longueurCarte faire
5:         listeDalles[I,J].dessiner
6:
7:     fin pour
8:
9: fin pour
```

4.5 Les événements et actions du jeu

Cette partie du programme a pour but de gérer les actions modifiant le jeu (par exemple les GameOver ou les blocs spéciaux). Cette partie sera gérée en 2 parties, d'abord les Evenements puis les Actions.

Evenements :
Ce module est constitué des éléments suivants :
Attributs :
caseDeclencheuse
action

Methodes :
void checkerEvenement(Joueur joueur)
boolean estSurCase(Joueur joueur)
void declencherAction()

C'est cette partie qui est interrogée par la boucle principale.

checkerEvenement(Joueur joueur)
C'est la fonction principale du module, elle se charge de lancer la vérification de coordonnées et de lancer l'Action associée en cas de réussite au test précédent.

Algorithme 12 AlgoPrincipal Action

```
1:
2: si estSurCase(joueur) = VRAI alors
3:         declencherAction()
4:
5: fin si
```

Elle va pour cela appeler les 2 autres méthodes du module :

estSurCase(Joueur joueur)
Renvoie VRAI si le joueur est présent sur la case, FAUX sinon. Techniquement cela consiste simplement en une comparaison des coordonnées X puis Y entre les attributs de Joueurs et ceux de l'Evenement (caseDeclencheuse).
Note : Pour gérer les évènements devant ÃŊtre déclenchés quelque soit la position du joueur. Nous avons décidé que si les coordonées de l'Evenement étaient (-1, -1), cette fonction ne faisait aucune comparaison et renvoyait VRAI.

declencherAction()
Cette méthode est chargée de lancer l'Action associée à l'Evenement. Elle va pour cela, lancer la méthode principale du module Action associé.

Actions :
Ce module est constitué des éléments suivants :
Methodes :
void realiserAction()
boolean verifierConditions()
void declencherAction()

Ce module est toujours appelé par Evenement.

realiserAction()
C'est la fonction principale du module, elle se charge de vérifier les conditions de déclenchement de l'Action, et le cas échéant de lancer l'Action proprement dites.

Algorithme 13 AlgoPrincipal Action

```
1:
2: si verifierConditions() = VRAI alors
3:         declencherAction()
4:
5: fin si
```

Elle va pour cela appeler les 2 autres méthodes du module :
verifierConditions()
Cette fonction est entièrement libre, elle doit juste renvoyer VRAI pour déclencher l'Action, FAUX sinon. Son but est de vérifier des éventuelles règles permettant de déclencher ou non l'Action.

declencherAction()
C'est la fonction qui va appliquer les changements visibles sur le jeu liés à cette Action.

Première partie

Mise en pratique en C#

Programmation sur XBOX 360

Ce chapitre permet de présenter les outils qui nous ont servis pour programmer notre jeu.

5.1 XNA

XNA est un framework développé par Microsoft facilitant le dévellopement de jeux vidéo. Basé sur le framework .NET, il n'est utilisable que dans le langage C# . L'objectif de ce framework est d'apporter une simplification dans l'écriture de code en fournissant des bibliothèques de fonctions pour le programmeur. Grace à cela on s'abstrait de l'aspect matériel, puisque à la compilation, c'est le type de la cible qui déterminera comment XNA doit agir.
XNA permet donc de créer des jeux pour différentes plate-formes qu'elles que : PC, Xbox360 et Windows Phone. Grace au framework, un projet XNA peut ÃŤÃÿÎľtre compilé, sans modification (ou peu), vers toute les plate-formes.

5.2 Visual Studio

Visual Studio est un environement de développement intégré (IDE) de Microsoft. Il permet d'utiliser XNA et contient tous les outils de conception, débugage multi-plateforme et de déploiment nécéssaire. De plus étant disponible gratuitement pour les étudiants via le MSDNA, c'est cet IDE qui nous servira à coder le jeu.

5.3 Le langage C#

Le langage C#, prononcé C sharp, est un langage de programmation orienté objet à typage fort. Le typage fort indique que les types de données employés décrivent correctement les données manipulées.
Développé par Microsoft et apparu en 2001, ces objectifs sont d'être un langage simple, moderne, généraliste et orienté objet. Il est proche du Java mais sa synthaxe reste semblable aux langage C++ et C.
Il est indispensable d'utiliser ce langage pour la programmation de la Xbox360 car c'est le seul autorisé par XNA. De plus, XNA étant disponible seulement pour le C#, il aurait été inpensable de coder le jeu en C pour la Xbox360 sans les librairies associées.

Composition d'un jeu XNA

Le domaine des jeux vidéo n'étant pas un domaine nouveau, il est très vite apparu des structures de gestion de jeu. Afin de faciliter le développement d'un jeu, les développeurs de XNA ont intégré nativement le projet de jeu XNA, des méthodes et structures d'éxécution adaptées. Cela donne plusieurs méthodes :

– Initialize(), qui permet d'initialiser les variables de l'écran de jeu
– LoadContent(), qui permet de de charger du contenu qui sera utilisé dans le jeu
– UnloadContent(), qui permet de décharger les objets pris lors du Load
– Update(), qui permet de gérer la logique de jeu.
– Draw(), qui permet de déssiner à l'écran le jeu. Moins souvent appelé que Update, il peut être exécuté au maximum 60 fois par secondes (c'est le framerate maximum par défaut pour un jeu Xbox360)

Toutes ces méthodes sont appelées automatiquement lors du chargement du jeu et de son exécution. Lors d'un lancement, c'est d'abord la méthode Initialisation qui est lancé en premier puis Load. Dans le cycle classique d'un jeu, c'est l'exécution d'un cycle Update puis Draw qui s'enchaine. Si le Draw prend trop de temps, il est possible que le programme décide de lui-même d'exécuter deux fois à la suite l'Update, attendant que l'instance de la méthode Draw en cours d'exécution ait fini son exécution.

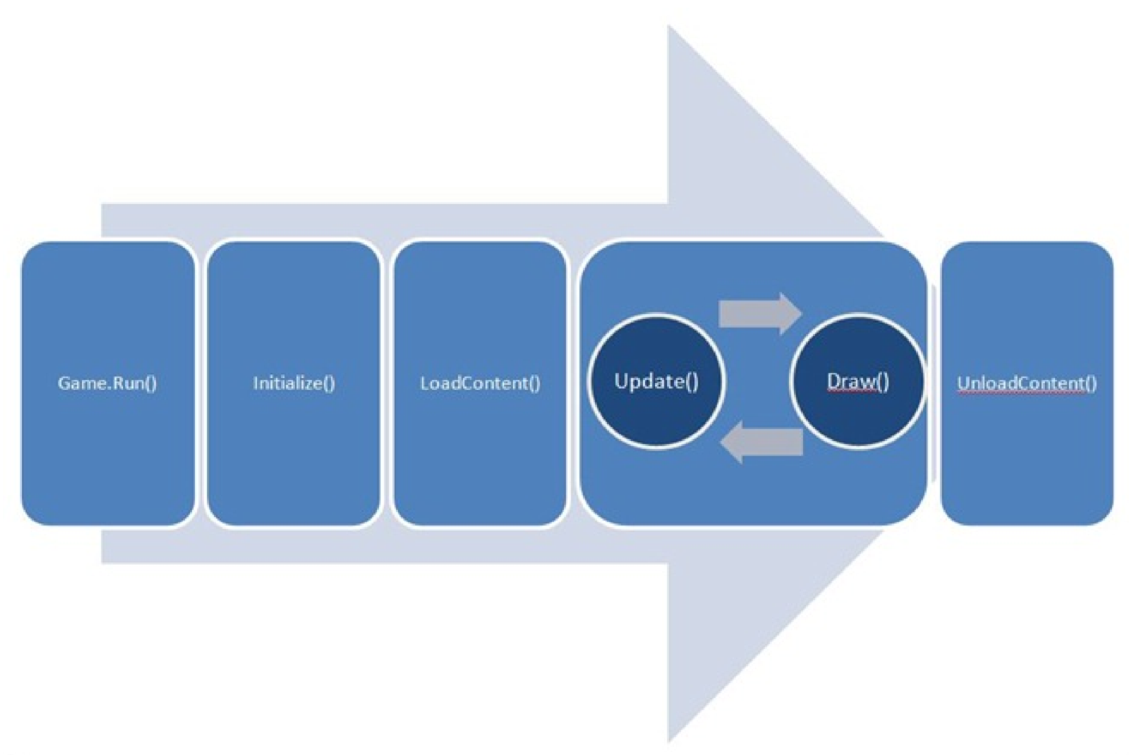

Figure 6.1 – Schéma de déroulement d'un jeu XNA

Le Game State Management

7.1 Principe

Dans la conception d'un jeu vidééo, il arrive un moment où la gestion des éécrans de jeu devient problèmatique. Il est donc nécessaire de normaliser les écrans de jeu et de faire intervenir un système de gestion permettant de donner la main à un écran ou à un autre suivant les état du jeu ou de la navigation dans ceux-ci.
Afin de sensibiliser les dévellopeurs à ce concept, Microsoft a mis en ligne sur le site de XNA un modèle de ce type, apportant un système complet de gestion d'éécran de jeu avec un menu graphique intégré. Ce modéle permet de créer des écrans de jeu normalisé et de répondre au aux besoins du système de gestion : Le ScreenManager.
Lancé au début du programme, il permet d'afficher l'écran de menu, composé du menu et du fond. Suivant les actions, d'autres écrans sont chargés et d'autres supprimés automatiquement. Il devient donc trivial de créer des chemins de navigation entre éécrans de menus et entre éécrans de jeu.
Il utilise des tests internes afin de déterminer si un écran a le droit d'être mis a jour ou dessiné (exécution du update et draw).

7.2 La classe BloxorzNiveauScreen

C'est une classe qui hérite de la classe GameScreen le type par défaut d'un écran de jeu et qui représentera notre jeu. Il permet de jouer à un niveau. Ensuite si le joueur fini le niveau il demandera au ScreenManager de lancer une nouvelle instance de BloxorzNiveauScreen avec de nouveaux paramètres, supprimant l'ancien écran par la même occasion. C'est le même principe si le joueur perd le niveau, une nouvelle instance de BloxorzNiveauScreen sera ajouté au ScreenManager avec les paramètres de départ de l'instance en cours et le nombre de tentative incrémenté.
Pour l'exécution, dans la fonction Update() sera placé la logique de jeu et de gestion des mouvements.
C'est dans la fonction Draw() que tout le jeu sera déssiné. Pour simplifier ces opérations, chaque entitée 3D visible dans le jeu posséde un fonction Draw() qui permet en fonction de ces coordonnées propres de s'afficher à l'écran.

La 2D et la 3D dans XNA

Pour nous, le passage du jeu de la version flash à la version Xbox 360 devait se faire par l'intégration de la 3D. Le framework XNA permet par des fonctions simples, de charger, modifier et afficher du contenu 3D. Pour l'affichage d'informations à l'écran, en 2D, d'autres méthodes nous sont proposées. Nous allons donc voir dans cette partie ce qui à été mis en oeuvre pour le jeu.

8.1 La 2D

Le dessin à l'écran de texte ou d'image ce fait par un SpriteBatch, qui est une instance d'un objet permettant de fournir, grâce à la création d'une liaison à l'initialisation de celui-ci avec la carte graphique, toutes les informations necéssaire à la carte graphique pour déssiner les actions demandées. Voici donc la déclaration d'un SpriteBatch et son initialisation avec la carte graphique :

```
SpriteBatch spriteBatch;
spriteBatch = new SpriteBatch(ScreenManager.GraphicsDevice);
```

Pour pouvoir effectuer des actions avec le SpriteBatch voici ce qu'il faut faire :

```
spritebatch.Begin();
...Nos actions pour le spritebatch...
spriteBatch.End();
```

De cette façon le SpriteBatch prend en compte les actions effectuées dessus. Notre SpriteBatch pouvant afficher du texte et des images, ils faut donc savoir comment les insérer dans notre jeu. Les images sont chargées dans un Texture2D de la faọn suivante :

```
Texture 2D imagesTuto= content.Load<Texture2D>("tutorial_1");
```

Pour le texte, c'est la police utilisée pour dessiner le texte qui est important. Une police sous XNA est représentée par un SpriteFont. Elle ce déclare et de charge de la façon suivante :

```
SpriteFont gameFont;
gameFont = content.Load<SpriteFont>("soulmission");
```

En général, pour chaque content.load, l'objet qui doit être chargé doit être ajouté au content du projet. Le content est disponible dans la barre de navigation du projet sous VisualStudio.
Vous avons par la suite utilisé deux méthode de SpriteBatch : l'affichage d'image et l'affichage de texte. L'affichage d'image nous a servis pour les images du tutoriel et pour indiquer au joueur le cube selectionné lorsque le cube est séparé en deux parties. Le texte a permis d'afficher des informations pratiques lors du déroulement du jeu ou lors du chargement d'un niveau. L'affichage d'une image ce fait par exemple de la faọn suivante :

```
spriteBatch.Draw(nom\_image, new Rectangle(Xd\'epart,Yd\'epart,Xfin,Yfin),
new Color(r, g, b));
```

Le nom_image correspond à une texture 2D. Le Rectangle à une variable Rectangle qui représente la position relative de l'image par rapport à l'écran sachant que l'image s'adaptera directement à la taille du rectange. Le Color à une variable de type Color qui prend 3 variables en entrée : le r (valeur de rouge), le g (valeur de vert),le b (valeur de bleu) toutes entre 0 et 255. L'affichage d'un texte se fait par exemple de la faņon suivante :

```
spriteBatch.DrawString(font, string.Format("TENTATIVES : {0}",
bloxorzGame.nombreTentative),textPosition, color);
```

Le font représente le SpriteFont qui sera utilisé. Le string.format permet permet d'effectuer une opération similaire au printf en C sauf que la sortie ne sera pas l'écran mais elle sera directement utilisée et permet de convertir une chaîne avec des arguments en un string (qui est le type du paramètre demandé). Le textposition est une variable de type Vector2D qui comporte deux coordonnées : X et Y. Le Color est une variable de type Color.

8.2 La 3D

8.2.1 La caméra 3D

La caméra permet de définir le point de vue qui sera présenté à l'écran lors de l'exécution du jeu. Elle est représentée par trois variables qui sont des matrices. Ces matrices seront par la suite utilisées à chaque fois que un objet 3D devra être déssiné. Ces trois matrices sont :

- World, qui définit le monde dans lequel se place l'object 3D, c'est la seule variable qui sera modifié directement lors d'un appel de dessin d'un objet 3D en fonction des paramètres de placement, de rotation ou d'échelle de l'objet
- View, permet de positionner la caméra dans l'espace 3D
- Projection, permet de définir les paramètres de la caméra comme le ratio la profondeur de champ ou la ligne d'horizon.

Le type Matrix qui compose chacune des variables décrites ici est une matrice 4x4. Dans l'initialisation d'un niveau, nous avons une fonction permettant de créer au début avant le lancement du jeu ces trois matrices, elle s ?appelle CreateCamera() ; .

8.2.2 Le cube 3D

Pour la partie 3D, nous avons d'abord penser à dessiner directement nos objets polygone par polygone (un polygone étant composé de trois sommets), en partant du fait que nos objects sont simples (forme carrée). Cependant après quelques manipulations, il se trouve que c'est très difficile à manipuler. Les opérations de translations et de rotations ne sont pas natives.
Le framework XNA propose dans sa palette de fonctionnalités, des méthodes pour charger manipuler et afficher des objets 3D. C'est donc de cette faņon que nous avons procédé pour les objects 3D.

Il y a pour l'instant un seul objet 3D utilisé dans notre jeu, cet objet est un cube conņu de faņon basique sous Blender, un logiciel de conception 3D gratuit. Cet objet à été exporté dans un fichier .fbx afin d'être chargé dans notre jeu. Il sert à représenter le joueur et les dalles. La manipulation des paramétres permet Le chargement du modèle 3D se fait de la faņon suivante :

```
Model Cube3d;
Cube3d = content.Load<Model>("cube");
```

Notre cube en 3D est donc stocké et exploitable dans une variable Model.
Lorsque un cube doit être dessiné, voici un exemple de ce qui doit être fait :

```
            foreach (ModelMesh mesh in game.Cube3d.Meshes)
            {
                foreach (BasicEffect effect in mesh.Effects)
                {
                    effect.EnableDefaultLighting();
                    effect.AmbientLightColor = Color.Orange.ToVector3();
                    effect.GraphicsDevice.DepthStencilState = DepthStencilState.Default;
                    effect.World = Matrix.Identity * Matrix.CreateScale(0.5f, 0.125f, 0.5f)
 * Matrix.CreateTranslation(
new Vector3(coordonnees.x, coordonnees.y, coordonnees.z));
                    effect.View = game.View;
                    effect.Projection = game.Projection;
                }
                mesh.Draw();
            }
```

On va donc revoir ce code en détail pour comprendre comment cela fonctionne. Dans notre exemple qui est le code d'une fonction Draw d'un dalle, le game.cube3d représente le Model de notre cube 3D chargé en mémoire dans le programme principal du niveau.
Pour chaque mesh (ensemble de polygones) de notre objet 3D pour chaque effect appartenant au mesh courant, on effectue plusieurs opérations. Dans l'ordre

- On active la lumière par défaut
- On Définit une couleur pour la lumière ambiante, ce qui permet de changer la couleur de l'objet
- On définit le DepthStencilState de l'objet à son état par défaut (actif), cela permet d'avoir la notion de profondeur lors du dessin et d'éviter que les polygones ne se chevauchent, si on le fait ici c'est parce que l'utilisation d'un SpriteBatch désactive cette fonctionnalité lors de son utilisation.
- On définit le monde dans lequel l'objet evolue en réglant la taille de l'objet (CreateScale)et sa translation (CreateTranslation) par rapport au point d'origine du monde 3D du jeu.
- On récupère la vue de la camera définie dans le programme principal
- On récupère la projection définie dans le programme principal

L'autre fonction qui est utilisé dans le jeu pour la matrice World est Matrix.CreateRotation(new Vector3(x,y,z)) qui en recevant des variables de type float pour x, y et z permet de créer des rotations suivants les axes x, y ou z.
Ce sont les bases pour l'affichage d'un objet en 3D. Nos fonctions de dessins d'objet 3D est basé sur cette structure.

Interaction homme-manette et le multimédia

Grâce au framework XNA, des fonctionnalitées nous sont fournis pour simplifier la mise en place de ces mécanismes.

9.0.3 Interactions avec la manette

Le Game State Management attribue à tout les érans de la classe GameScreen, une méthode afin de centraliser la lecture de l'état des manettes. Cette méthode s'appelle HandleInput et prend en paramètre un InputState. Cette méthode de BloxorzNiveauScreen est appellée automatiquement par le GSM lorsque l'écran fait partie de sa liste d'écran à mettre à jour et qu'il a la main sur le jeu. Cela permet d'appeler la méthode seulement si le jeu est en cours afin d'ééviter de répercuter les évenements manette dans le jeu si celui-ci est sur pause. D'autres mééthodes sont présentes pour savoir si un nouveau bouton est enfoncé sur une manette ou si le bouton pause est enfoncé. L'appel à une des ces méthodes nécéessite toujours le numéro de la manette ou du clavier (considéré comme le numéro du joueur).
C'est donc dans cette partie que notre jeu réécupèére tous les éévéénements de la manette. Elle permet dans un premier temps, de vérifier si le jeu doit se mettre en pause si le bouton pause est appuyé ou si la manette est déconnectée de la console. Si ce n'est pas le cas, elle lance la mise à jour des événements par la fonction InputUpdate().

```
private void updateInput(InputState input)
{
    inputEvenement.up = inputActionUp(input);
    inputEvenement.down = inputActionDown(input);
    inputEvenement.left = inputActionLeft(input);
    inputEvenement.right = inputActionRight(input);
    inputEvenement.change = inputActionChange(input);
}
```

Voici une des fonctions inputActionxxx(input) qui renvoie directement le résultat de la fonction de lecture de l'état d'un bouton.

```
private bool inputActionUp(InputState input)
{
    PlayerIndex temp;
    return (input.IsKeyPress(Keys.Up, PlayerIndex.One, out temp)
|| input.IsButtonPress(Buttons.DPadUp, PlayerIndex.One, out temp));
}
```

Le stockage des événements ce fait dans une structure composée de variable booléennes permettant d'indiquer si des événements ont lieu ou non.

```
struct InputEvenement
{
    public bool up;
```

```
            public bool down;
            public bool left;
            public bool right;
            public bool change;
        }
        InputEvenement inputEvenement;
```

Afin d'obtenir un meilleur gameplay, deux nouvelles méthodes ont été implémentèes dans la classe InputState : IsKeyPress et IsButtonPress. La première permet de savoir si un bouton du clavier est pressé, la seconde si un bouton sur la manette est pressé.
Pour chaque bouton auquel on va associer un événement, on effectue une lecture de celui-ci, soit par IsButtonPress, soit par IsNewButtonPress. Cette structure d'événements sera réinitialisée à zéro dans l'update du jeu aprés avoir été pris en compte.

9.0.4 Le multimèdia

Pour le projet seul des fichiers audio ont été utlisés comme multimédia. Deux types de sons ont été utilisés : les types SoundEffect et Song.
Tous deux fournis par XNA, le premier doit êêtre un fichier audio au format WAV (fichier son non compréssé) qui permet de jouer des bruits trèés courts. C'est de cette façéon que le bruit des cliquetis du joueur sont joués. Le type SoundEffect charge le fichier son dans la mémoire vive de la console afin qu'il soit joué de façéon fluide. On effectue le chargement d'un joueur dans un SoundEffect par :

```
public SoundEffect clic;
clic = content.Load<SoundEffect>("clic");
```

Il faut avant cela avoir ajouté le fichier dans la section content du projet afin d'êêtre reconnu par XNA. Pour jouer le son on fait :

```
clic.Play();
```

Le second type, Song, peut êêtre un fichier compréssé en MP3 par exemple, il est décodé par un composant intégré XNA, le MediaPlayer, et sa lecture pendant le jeu s'effectue sur le disque. Cela donne un temps de réponse plus long et la mise en place du décodeur peut provoquer des ralentissements, surtout si beaucoup d'actions de lecture et d'arrêêt sont effectuées dans un laps de temps très court. Le chargement s'effectue de la manièère suivante :

```
Song song;
song = content.Load<Song>("CCCS");
```

Et les actions de bases sont les suivantes :

```
MediaPlayer.Play(song);
Mediaplayer.Pause();
MediaPlayer.Resume();
```

Les Actions

Afin de rendre plus génèérique l'implémentation des Actions, nous avons utilisé le concept objet d'Interface. Sans entrer dans les détails, cela permet de créer un "modèle" que les classes Action devront obligatoirement suivre.
Cette Interface, appelè e ActionJeu spécifie les 4 fonctions que doivent forcément être implémentèés :
void realiserAction(BloxorzNiveauScreen game) ;

/* Renvoie vrai si les conditions (facultatives) à la réalisation de l'action sont respectèes, faux sinon */
bool verifierConditions(BloxorzNiveauScreen game) ;

/* Code de l'action à proprement parler */
void declencherAction(BloxorzNiveauScreen game) ;

/* Gére l'animation de cet événement */
void updateAnimationAction(BloxorzNiveauScreen game) ;

Cette méthode a 2 avantages :
Premièérement, elle permet de gérer tous les Evenements avec le mêéme code dans la boucle principale, toutes les spécificités sont gérèés en interne par la classe.
Deuxiémement, cela permet de prévoir l'ajout de nouvelles Actions dans le futur, et cela sans avoir a toucher le code principal du jeu.

Liste des Actions

Voici les Actions implémentés dans le jeu actuellement :

Action	Nom de le Classe	Commentaire
Pont	ActionCreerPont	Crèer un pont
Bloc Orange	ActionBlocOrange	Tombe sur le bloc est debout dessus
S ?parateur	ActionSeparateur	Sèpare le bloc en 2 cubes
FinNiveau	ActionFinNiveau	Gère la fin du Niveau
GameOver	ActionGameOver	Gère les GameOver

Lecture de niveau

Nous avons ajouté la possibilité de charger un Niveau depuis un fichier, cette fonctionnalité a été ajouté sur la fin du projet et pourrait donc être améliorée, cependant, l'algorithme principal est codé et fonctionne correctement.
Cet algorithme est trés similaire à l'algorithme de chargement de Niveau classique. Quelques diffèérences sont tout de mêéme à noter :
L'ouverture du fichier :
Nous avons décidé (dans un soucis de délais) que le programme ne permettrais d'ouvrir un seul Niveau de cette maniére, et que son nom serait fixé dans le code source (fichier.txt ici).
Cette ouverture se fait comme ceci :

Listing 12.1 – Ouverture du Fichier

```
string fichierNiveau = string.Format("Content/niveau.txt");
Stream fileStream = TitleContainer.OpenStream(fichierNiveau);
StreamReader reader = new StreamReader(fileStream);
```

Nous avons donc cré un Reader, gréce auquel on peut récupérer une ligne du fichier de cette façon : monReader.ReadLine()
Il reste donc à lire le Niveau dans le fichier, nous avons pour cela, défini une norme de fichiers :

Listing 12.2 – Ouverture du Fichier

```
Niveau
largeurCarte
LongueurCarte
DEBUT_DALLES
Xdalle1
Ydalle1
Xdalle2
Ydalle2
FIN_DALLES
nombreEvenements
DEBUT_ACTIONS
XcaseDeclencheuse1
YcaseDeclencheuse1
NomAction1
[Optionnel : param\`etres Action1]
TypeEstSurCase1
XcaseDeclencheuse2
YcaseDeclencheuse2
NomAction2
[Optionnel : param\'etres␣Action2]
TypeEstSurCase2
FIN_ACTIONS
```

```
XDepart
YDepart
```

La seule spécificité se trouve dans le stockage des Evenements. En effet, le nombre de paramétres d'un Evenement dépend du type de l'Action lié à celui-ci. Il a du fallu gérer cela avec une structure en switch sur la valeur "NomAction".
Par manque de temps, nous n'avons pas géré les erreurs dues à un fichier non conforme.

Déploiement sur Xbox360

Une fois notre projet terminé, il faut pouvoir tester notre jeu sur Xbox360. Pour cela il est nécessaire d'avoir un compte Xbox Live qui permet à la console d'utiliser les ressources microsoft. De plus il faut avoir un compte XNA dévelopeur qu'il faut lier au compte Xbox Live afin de créer un lien entre les deux.
Ensuite la console doit être reconnue par le framework Xbox sur la machine de dévelloppement. On lance XNA devices puis ädd devicesët on rentre le nom et le code d'authentification de la Xbox. Ce code est donnée par un logiciel qui doit être installé ẌNA Game Developper tools, une fois le compte Xbox live et XNA dévelopeur associés. Celui-ci lorsqu'il est lancé, il permet d'obtenir un code permettant à un PC avec XNA de pouvoir ajouter la Xbox. Cette liaison faite, on peut directement dans visual studio effectuer des actions de débbugage et d'exécution directement sur la console. Cette démarche nous a été présenté par Mickael ROUSSEAU, qui à déja travaillé sur des projets de dévellopement sur Xbox360.

Résultats

Voici quelques captures d'écran réalisés à partir du jeu.

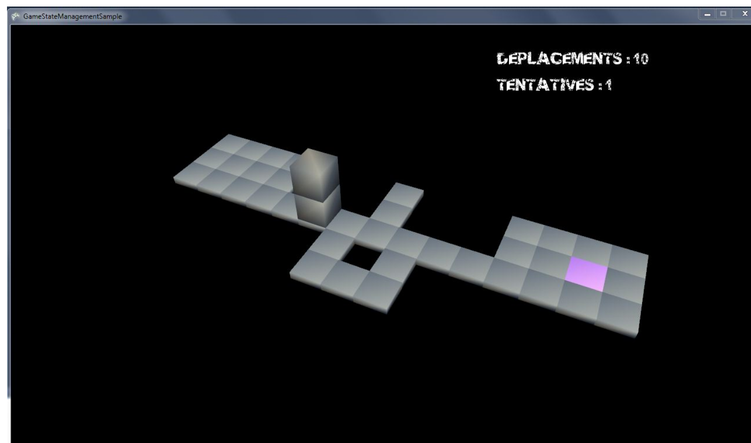

FIGURE 14.1 – Image du jeu : joueur en un morceau

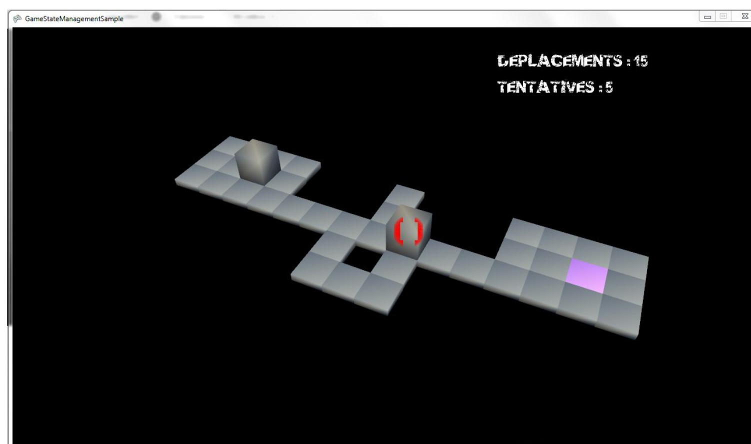

FIGURE 14.2 – Image du jeu :affichage du cube selectionné lorsque le joueur est en deux morceaux

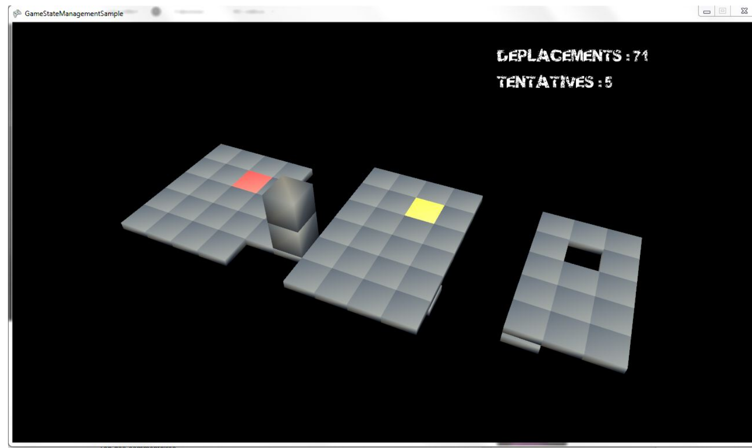

FIGURE 14.3 – Image du jeu : Visuel des dalles actions (rouge et jaune) et d'un pont crée par une action

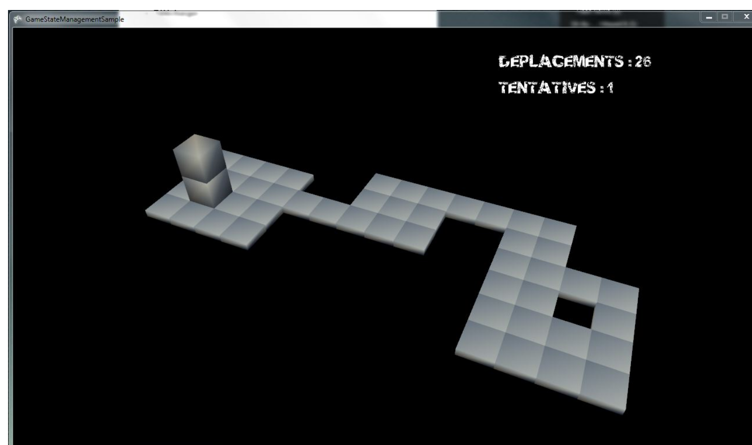

FIGURE 14.4 – Image du jeu : Un niveau classique

Les objectifs atteints

Nous avons réussi à réaliser la majorité des objectifs du projet :
- Le jeu est fonctionnel.
- Il tourne sur la XBOX 360.

15.1 Pour la suite

Des évolutions sont possibles pour ce projet :
- Il est tout a fait possible, avec notre structure de créer de nouveaux blocs avec de nouvelles interactions.
- Notre menu est tout a fait basique et il serait possible d'ajouter des réglages tels-que la musique.
- Il serait aussi possible de créer un système de "HighScore".
- Le système de chargements des niveaux est facilement modifiable pour se faire exclusivement via des fichiers, il faudra cependant rendre cette lecture plus sure et fiable sous risque de rendre le programme instable.
- Un chargement des niveaux par fichiers rend possible la création d'un éditeur de niveaux.

Conclusion

Nous sommes assez fiers de ce projet car nous avons réussi à le mener à bien en 3D comme nous le souhaitions.
Le fait de créer un jeu était très intéressant et gratifiant, le coté ludique du jeu apporte vraiment de la motivation à travailler sur le sujet.
Ce projet nous a permis de mettre en application de nombreux concepts vus en cours durant l'année, cela rend la formation plus concrête.

Développement d'un jeu Bloxorz sur Xbox360

Département Informatique
3e année
2011 - 2012

Rapport de projet d'Algorithmique / Langage C

Résumé : Description en français

Mots clefs : Mots clés français

Abstract: Description en anglais

Keywords: Mots clés en anglais

Encadrants
Carl ESSWEIN
carl.esswein@univ-tours.fr
Jean-Louis BOUQUARD
jean-louis.bouquard@univ-tours.fr

étudiants
Benoit BELZ
benoit.belz@etu.univ-tours.fr
Fabien FARIN
fabien.farin@gmail.com

Université François-Rabelais, Tours

DI3 2011 - 2012