Travail de maturité 2008

Création d'un jeu vidéo utilisant un moteur physique

David Perrenoud

Répondant : Gérard Valzino

Gymnase de Beaulieu

Lausanne

Remerciements

Je tiens à remercier tous les camarades qui ont relu ce rapport et testé le jeu, ainsi que Monsieur Valzino pour son soutien tout au long du travail.

Table des matières

Introduction

Il est difficile de situer précisément le début de l'histoire du jeu vidéo. Toutefois, le premier à obtenir un succès commercial fut *Pong*, en 1972. *Pong* consiste en une simulation simpliste de tennis de table en 2D. Le joueur doit déplacer verticalement une palette et dispute le match contre la machine ou contre un autre joueur. Si un des joueurs ne réussit pas à renvoyer la balle, l'adversaire gagne un point.

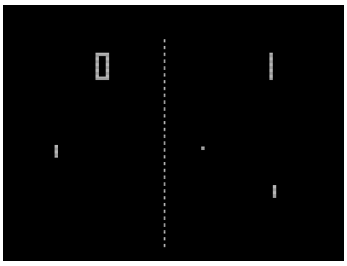


FIG. 1 – *Pong* : où tout a commencé...

Depuis *Pong*, les progrès se sont fait tant au niveau matériel, avec des ordinateurs et des consoles toujours plus rapides et miniaturisés, qu'au niveau logiciel, avec des manières de jouer novatrices et un réalisme toujours plus saisissant. Ces deux aspects se rejoignent dans l'utilisation d'un moteur physique : il nécessite d'intenses calculs et permet d'inventer une jouabilité qui n'existait pas auparavant.

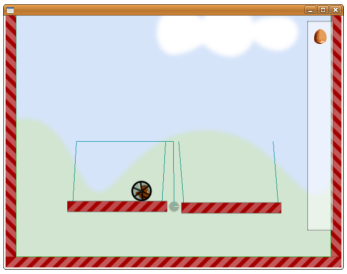


FIG. 2 – *JAMP* : le projet de ce travail de maturité.

Lors de la décision des thèmes des travaux de maturité, je voulais à la fois gérer un projet et utiliser mes connaissances en informatique et en physique. C'est pourquoi j'ai décidé de créer *JAMP*, qui signifie **J**eu **A**vec un **M**oteur **P**hysique.

La création d'un jeu est un parcours long et sinueux. Il faut principalement écrire du code, mais aussi gérer l'emploi du temps, définir des objectifs, créer des graphismes, inventer des règles, concevoir un niveau, ajouter des sons, etc.

Ce rapport est découpé en trois parties : la première décrit le jeu et son fonctionnement, la seconde raconte le déroulement du travail et la troisième explique le code du jeu de manière technique.

Le jeu sera disponible dès janvier 2009 à l'adresse : `http://perre.noud.ch/jamp`

Chapitre 1

Le jeu

1.1 Règles

Le jeu est en 2D. Le joueur dirige un écureuil dans une roue avec le clavier et peut ajouter des objets avec la souris. Le but du jeu est d'arriver à la noisette qui se situe à la fin de parcours. Ce parcours est composé de plateformes fixes ou mobiles, ainsi que d'objets divers. La roue de l'écureuil permet au joueur de se mouvoir facilement dans le niveau. Le choix de l'animal, plutôt que d'un élément neutre, vise à ajouter une touche de sympathie au jeu. Il en est de même pour les graphismes très colorés.

1.2 Choix particuliers

Quatre plateformes principales limitent l'espace du joueur : une en haut, une en bas, une à gauche et une à droite. Dans certains jeux, il y a *Game Over* lorsque le personnage sort de l'écran. Dans le cas présent, ce n'est pas possible puisque le personnage ne peut dépasser ces quatre plateformes. Bien que cela diminue la difficulté et la nervosité du jeu, cela le rend moins frustrant et plus propice à la réflexion.

L'intérêt du jeu provient des objets mis à disposition et de la complexité du niveau. De plus, le cheminement du joueur est totalement libre. Ce dernier se retrouvera souvent à expérimenter des choses sans rapport avec le niveau, simplement pour leur caractère ludique.

Une grande partie des jeux actuels, y compris celui-ci, ont tendance à donner une grande liberté au joueur. Celle-ci est obtenue grâce au moteur physique : il permet d'éviter que chaque action engendre un résultat défini précisément par le programmeur, ce qui peut donner des résultats étonnants.

1.3 Moteur physique

Un moteur physique permet de simuler le mouvement d'objets comme s'ils bougeaient dans le monde réel, en appliquant les lois de la mécanique classique.

J'ai choisi de ne pas créer un moteur physique moi-même. Tout d'abord, parce que la programmation d'un moteur de ce type m'aurait pris autant de temps que la création du jeu, pour aboutir à un résultat moins bon. Ensuite, parce qu'il nécessite des connaissances poussées en programmation, ainsi que de bonnes bases en mathématiques et en physique. Enfin, car il existe déjà un très bon moteur utilisable librement, nommé *Box2D*[1].

Le choix de *Box2D* s'est basé sur ses fonctionnalités, ses performances et sa stabilité. Un autre concurrent libre était aussi disponible, *Chipmunk*[2], mais il disposait d'une physique moins stable, de fonctionnalités moins avancées et d'un code moins clair.

1.3.1 Objets et formes

Il y a deux types d'objets dans *Box2D* : les objets statiques et les objets dynamiques. Les objets statiques, comme leur nom l'indique, ne peuvent pas se mouvoir. Ils sont donc très stables et n'entrent pas en collision entre eux. Dans le jeu, ils forment certaines plateformes. Les objets dynamiques composent tout le reste : les plateformes mouvantes, la roue, l'écureuil, les éléments à ajouter avec la souris, etc. Chaque objet dynamique possède, entre autres, une vitesse en $[m \cdot s^{-1}]$, une position en $([m], [m])$ et une ou plusieurs formes. Chaque forme possède un coefficient de friction, un coefficient de rebond et une densité en $[kg \cdot m^{-2}]$, qui permettent de calculer la masse de l'objet.

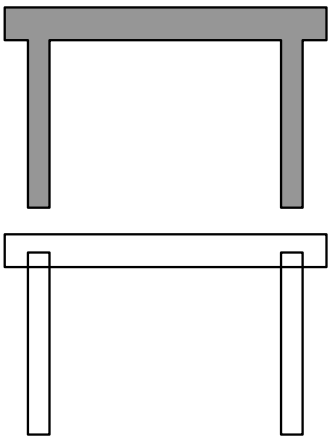


FIG. 3 – En haut : l'objet table. En bas : les trois formes qui le composent.

D'un point de vue technique, *Box2D* permet uniquement de créer des objets rigides. C'est-à-dire que la distance entre deux morceaux de matière reste constante, qu'aucune déformation n'est possible. Par exemple, pour un moteur physique 2D, une table est un objet composé de trois formes de type polygonal : une planche et deux pieds. Les deux pieds restent fixes par rapport à la planche. La complexité du code et les calculs d'un moteur physique

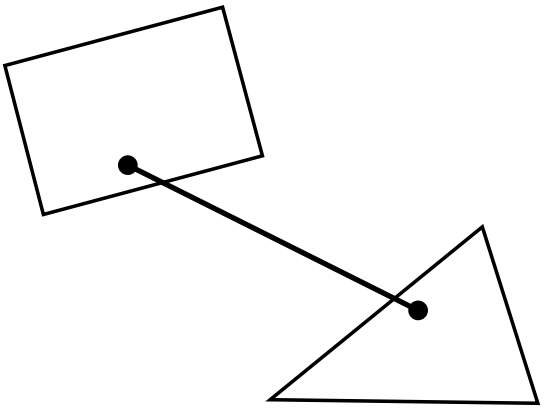
[1] `http://www.box2d.org/`
[2] `http://wiki.slembcke.net/main/published/Chipmunk`

d'objets rigides sont bien moindres comparés à ceux d'un moteur d'objets mous, ce qui a orienté le choix du créateur de *Box2D*. Il est tout de même prévu dans les fonctionnalités à venir de *Box2D* d'ajouter une gestion des fluides.

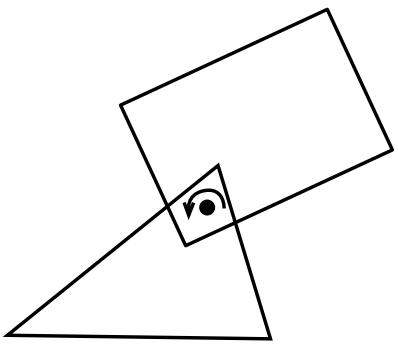
1.3.2 Joints

En 2D, un objet a trois degrés de liberté : une position dans le plan x, y et un angle de rotation θ. Un joint contraint le mouvement d'un objet en fonction d'un autre objet. Il lui enlève ainsi un ou plusieurs degrés de liberté. On peut utiliser les joints pour créer des pantins, des balances, des bicyclettes, etc. Une limite et un moteur peuvent être ajoutés à certains joints (voir plus bas). Par défaut, il existe plusieurs types de joints disponibles dans *Box2D* dont voici leurs définitions :

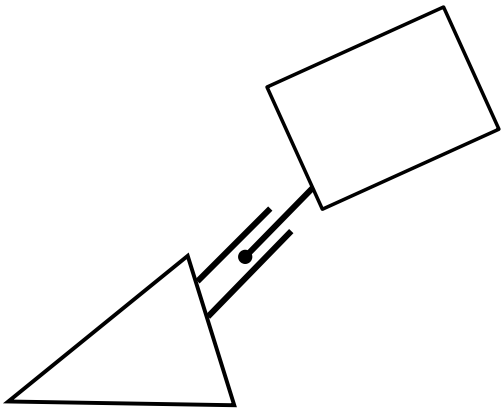
– Le joint de type *distance* oblige deux points situés sur deux objets différents à rester à distance constante.



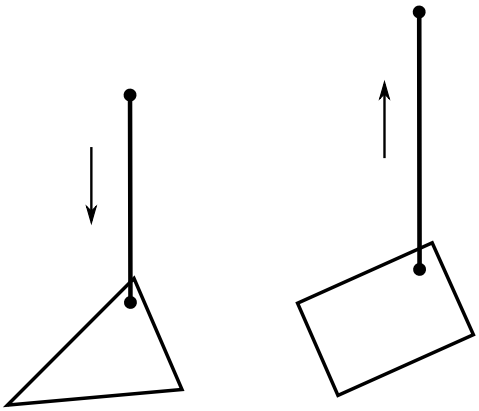
– Le joint de type *révolution* oblige deux objets à tourner autour d'un point commun. On peut définir une limite afin de, par exemple, restreindre une jambe à 90° de rotation par rapport au corps. Aussi, un moteur peut être ajouté pour, par exemple, modifier l'angle d'un coude.



– Le joint de type *prismatique* oblige deux objets à se déplacer selon un axe relatif. Il interdit donc la rotation entre les deux objets.



– Le joint de type *poulie* oblige un objet à monter si l'autre descend et réciproquement.



1.4 Moteur graphique

Le moteur graphique comporte schématiquement trois parties : les textures, la caméra et le processus de rendu. Une texture est une image assignée à un objet afin de pouvoir l'afficher à l'écran. Elle a plusieurs paramètres : sa taille à l'écran, sa position, sa symétrie d'axe horizontal ou d'axe vertical, sa colorisation et son coefficient de transparence.

La caméra voit un certain espace du jeu tout en se déplaçant avec le joueur. Elle se charge aussi de convertir les unités de l'écran et du jeu. En effet, contrairement aux coordonnées du jeu, celles de l'écran restent toujours fixes : l'origine se situe toujours en bas à gauche de la fenêtre et l'unité reste le pixel (fig. 4, page 8). Techniquement, la caméra stocke la position et le zoom du point de vue et envoie ces deux paramètres au processus de rendu.

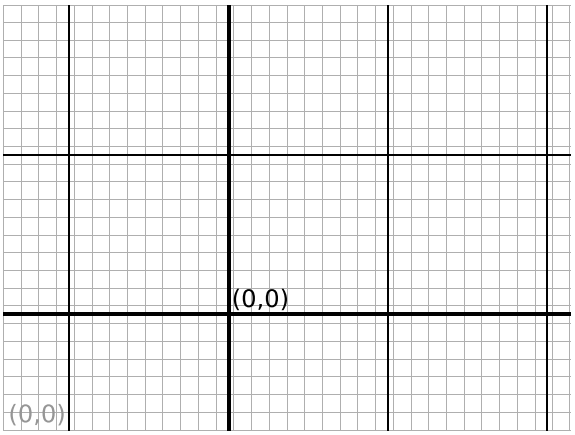


FIG. 4 – En gris : coordonnées de l'écran. En noir : coordonnées du jeu.

Dès le lancement du jeu, le processus de rendu crée la fenêtre principale. La taille de la fenêtre est envoyée à la caméra afin de régler le zoom en conséquence. Chaque fois qu'il faut recharger l'image, le processus de rendu demande le point d'origine et l'unité à la caméra. Relativement à ces coordonnées, il affiche les textures dans leur ordre inverse de proximité (de l'arrière-plan à l'avant-plan). La plus proche doit être rendue en dernier, car elles s'empilent les unes sur les autres.

1.5 Périphériques d'entrée

1.5.1 Clavier

Chaque touche du clavier dispose de trois états possibles : enfoncé, retenu et relâché. Dans le jeu, les touches gauche et droite commandent la rotation du personnage. La touche gauche ajoute du couple dans le sens contraire des aiguilles d'une montre tandis que la touche droite le fait dans le sens inverse. La touche bas pousse le personnage vers le bas.

La touche haut produit un saut du personnage. Ce saut est autorisé uniquement si le personnage touche le sol. Pour vérifier cette condition, un capteur a été ajouté au bas de la roue (fig. 5).

FIG. 5 – En vert : le capteur qui autorise le saut.

1.5.2 Souris

Tout comme le clavier, chaque bouton de la souris dispose de trois états possibles : enfoncé, retenu et relâché. Dans le jeu, la souris permet d'ajouter des éléments par glisser-déposer. Les coordonnées de la souris sont converties par la caméra (voir la section Moteur graphique, page 7).

1.6 Son

Les éléments sonores du jeu se composent uniquement de quelques bruitages. Ils sont déclenchés par le moteur physique : lorsqu'il détecte une collision, il envoie un signal au processus du son. Ce dernier produit un son correspondant aux objets de la collision, selon une liste prédéfinie. Des morceaux sonores peuvent aussi être assignés aux actions de la souris et du clavier, ou à toute autre événement.

Chapitre 2

Déroulement du travail

Une approche fondée sur la pratique a été utilisée pour développer le jeu. J'ai ainsi cerné mes buts et mes impératifs en me lançant directement dans la programmation, avec l'écriture de petits programmes de test. Cette étape a nécessité d'apprendre le langage de programmation *C* en lisant un livre sur le sujet[1], en février 2008. Les premiers objectifs de ce travail ont été définis en avril 2008 et les tests ont duré jusqu'en août 2008. Ils m'ont appris qu'un moteur physique pouvait aisément simuler plus de 2000 objets circulaires à une fréquence de 60 images par seconde. J'ai ensuite réussi à charger et afficher des textures.

Au final, le code de ces tests, de même que les premiers outils utilisés pour le créer, n'ont pas été récupérés dans le jeu final. Cette étape m'a cependant permis de mettre mes connaissances en pratique dès le tout début de mon travail, ce qui m'est paru une bonne méthode d'apprentissage. J'ai fait cet apprentissage en autodidacte, procédant par essais et erreurs, avec comme résultat la nette satisfaction d'avoir découvert la matière par moi-même.

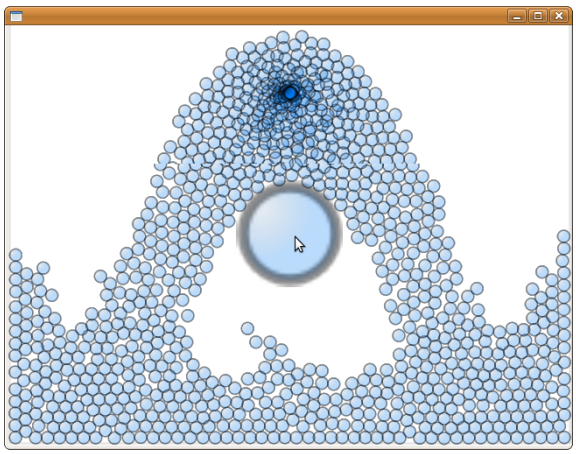


FIG. 6 – Test de simulation en temps réel de plus de 2 000 objets circulaires.

Chapitre 3

Le code

Mon jeu est écrit en *C++*. Il en est de même pour *Box2D* et le code sur lequel je me suis basé[2]. La compréhension de ce qui suit nécessite de bonnes connaissances de ce langage de programmation. Si vous ne savez pas ce qu'est le *C++*, inutile de continuer plus loin... Il existe tout de même des livres de vulgarisation sur le sujet, comme celui de Wikibooks[3]. Si le fonctionnement du moteur physique vous intéresse, lisez les explications de *Chris Hecker* sur la théorie des moteurs physiques d'objets rigides[4] et étudiez la documentation de *Box2D*[5] pour la pratique.

Pour facilitier la compréhension et par souci de concision, les exemples de code sont proposés en version simplifiée. Les caractères « `//` » signalent un commentaire, de la même manière qu'en *C++*. Pour une étude plus approfondie, le code source complet est disponible à l'adresse : `http://perre.noud.ch/jamp`. J'ai choisi de ne pas l'imprimer, car il occuperait plus de 70 pages.

3.1 Structure des fichiers

Le code est découpé en plusieurs fichiers. Chaque fichier contient une partie bien précise du jeu. Voici la liste de leurs fonctions, par ordre alphabétique :

cActorManager.cpp : crée les éléments du jeu et stocke leurs propriétés.

cCamera.cpp : stocke la position et le zoom de la caméra.

cDebugRender.cpp : affiche les formes physiques des éléments afin de faciliter le développement (désactivé dans le jeu final).

cDragBox.cpp : contient les adresses des éléments à ajouter dans le niveau à l'aide de la souris.

cGame.cpp : crée les classes principales du jeu, comme par exemple *cInput*, et la boucle d'événements (rechargement de l'image, événements de la souris, du clavier et de la fenêtre).

cGraphics.cpp : crée la fenêtre et affiche les textures.

cInput.cpp : stocke en mémoire l'état du clavier et de la souris.

cLevel.cpp : initialise l'environnement physique, crée le niveau et transmet certains paramètres au moteur physique.

cMusic.cpp : stocke en mémoire une musique.

cProcessManager.cpp : liste des processus à appeler à chaque rechargement de l'écran, c'est-à-dire 60 fois par seconde.

cSound.cpp : stocke en mémoire un bruitage.

cSoundManager.cpp : lit les bruitages et la musique.

cTextureManager.cpp : charge les textures et stocke leurs propriétés (transparence, rotation, symétrie, etc.)

globals.h : inclut les en-têtes des librairies externes utilisées (*OpenGL*, *GLU*, *SDL*, *SDL_image*[1], *SDL_mixer*[2], *Box2D*) et définit le chemin du dossier « images ».

3.2 Compilation

Une fois le code écrit, il faut le transformer en un fichier exécutable. Cette étape est décrite dans le fichier « Makefile » pour Linux et dans le fichier « CodeBlocks/-JAMP.cbp » pour Windows. Je présente ici uniquement la compilation pour Linux.

Le principal compilateur pour Linux est *GCC*[3]. Un programme *C++* trivial composé d'un seul fichier se compile avec la commande suivante :

```
g++ test.cpp -o test
```

Un programme plus complexe utilise un fichier « Makefile » qui exécute plusieurs commandes. Pour commencer, il compile un à un les fichiers « .cpp » en « .o » :

```
%.o:        %.cpp
    g++ $(CXXFLAGS) -c $< -o $@
```

Ensuite, il attache les fichiers « .o » ensemble, ainsi que les librairies, pour donner le binaire :

```
jamp:       $(FLOAT_OBJECTS)
    g++ -o $@ $^ -L$(PROJECT)/Source/Gen/float -lbox2d -lGL -lGLU↩
        -lSDL -lSDL_image -lSDL_mixer
```

[1] http://www.libsdl.org/projects/SDL_image/
[2] http://www.libsdl.org/projects/SDL_mixer/
[3] GNU Compiler Collection

Le fichier *Makefile* peut aussi contenir une procédure d'installation, qui crée les dossiers et copie les fichiers, et une procédure de désinstallation :

```
make-install-dirs:
    mkdir -p $(DESTDIR)$(BINDIR)
    mkdir -p $(DESTDIR)$(DATADIR)/jamp
    mkdir -p $(DESTDIR)$(DATADIR)/jamp/images

install:    make-install-dirs
    install -m 755 jamp $(DESTDIR)$(BINDIR)
    install -m 644 images/* $(DESTDIR)$(DATADIR)/jamp/images

uninstall:
    rm -f $(DESTDIR)$(BINDIR)/jamp
    rm -rf $(DESTDIR)$(DATADIR)/jamp
```

3.3 L'écureuil et sa roue

J'explique ici le fonctionnement de l'écureuil dans sa roue, car il est astucieux et j'en suis assez fier. Physiquement, l'écureuil devrait pousser la roue avec ses pieds pour la faire tourner. Le but d'un programmeur quand il fait un jeu est de rendre l'effet le plus réaliste possible, mais pas obligatoirement le plus proche de la réalité. L'utilisation d'un moteur physique ne change rien à la donne. Par exemple, il est conseillé d'utiliser une accélération gravitationnelle plus élevée que $9.81\ m \cdot s^{-2}$ si cela semble plus réaliste.

Dans le jeu, ce n'est pas l'écureuil qui pousse la roue, mais la roue qui déplace l'écureuil. Celui-ci est fixé à la roue à l'aide d'un joint de type révolution (voir page 6). Un couple inverse à celui de la roue lui est appliqué. Un coefficient de stabilisation angulaire (*angularDamping*) ralentit sa rotation. Enfin, ils se trouvent dans le même groupe de collision, donc ils ne s'entrechoquent pas. Cela est défini par le code suivant :

```
b2BodyDef ballDef;
ballDef.position.Set(x, y);              // positionnement
b2Body* ball = SetBody(world->CreateBody(&ballDef));
b2CircleDef ballShape;
ballShape.radius = 0.2f;                 // roue de 20 cm de rayon
ballShape.density = 1.0f;                // rapport masse / volume
ballShape.friction = 1.0f;               // coef. de friction
ballShape.filter.groupIndex = 1;         // groupe de collision 1
ball->CreateShape(&ballShape);
ball->SetMassFromShapes();               // calcul de la masse

b2BodyDef squirrelDef;
squirrelDef.position.Set(x, y - 0.05f); // positionnement
squirrelDef.angularDamping = 8.0f;      // stabilisation angulaire
b2Body* squirrel = world->CreateBody(&squirrelDef);
b2PolygonDef squirrelShape;
squirrelShape.SetAsBox(0.1f, 0.1f);      // rectangle 10x10 cm
```

```
squirrelShape.density = 1.0f;            // rapport masse / volume
squirrelShape.friction = 0.5f;           // coef. de friction
squirrelShape.filter.groupIndex = 1;     // groupe de collision 1
squirrel->CreateShape(&squirrelShape);
squirrel->SetMassFromShapes();           // calcul de la masse

b2RevoluteJointDef jd;
jd.Initialize(ball, squirrel, ball->GetWorldCenter());
world->CreateJoint(&jd);
```

3.4 Redimensionnement dynamique de texture (MIP mapping)

Le MIP mapping est une méthode de pré-calcul et d'optimisation des textures afin d'accélérer leur affichage et de réduire les artefacts. « MIP » est un acronyme latin de *multum in parvo*, littéralement « beaucoup dans un petit espace ». Le MIP mapping utilise la texture de départ pour créer des versions plus petites et déjà anti-aliasées. Ces versions réduites sont utilisées si leur précision est suffisante.

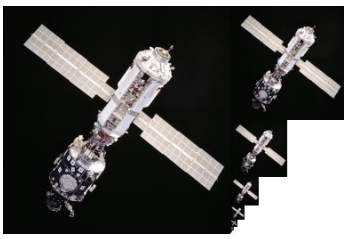


FIG. 7 – L'image de départ, à gauche, est utilisée pour créer des versions réduites.

Dans la pratique, j'ai utilisé la librairie *GLU*[4] afin de créer facilement les mipmaps. La première ligne de code génère les mipmaps tandis que la seconde définit la méthode de redimensionnement à utiliser si la texture est plus petite à l'écran que sa taille réelle.

```
gluBuild2DMipmaps(GL_TEXTURE_2D, mode, surface->w, surface->h, ↩
    mode, GL_UNSIGNED_BYTE, surface->pixels);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↩
    GL_LINEAR_MIPMAP_LINEAR);
```

[4]OpenGL Utility Library

3.5 Détection de collisions continues

La détection de collisions continues est une fonction de *Box2D* qui a fait son apparition en mars 2008, dans la version 2.0.0. Elle permet de résoudre le problème de l'effet tunnel. L'effet tunnel s'observe lorsque deux objets se déplacent rapidement. À l'aide de la position d'un objet à un instant donné, la simulation calcule sa position à l'instant suivant. Elle résout ensuite la collision si elle a lieu. Il peut arriver qu'un objet soit passé à travers l'autre si sa vitesse était suffisante pour qu'il l'ait dépassé complètement à l'instant suivant.

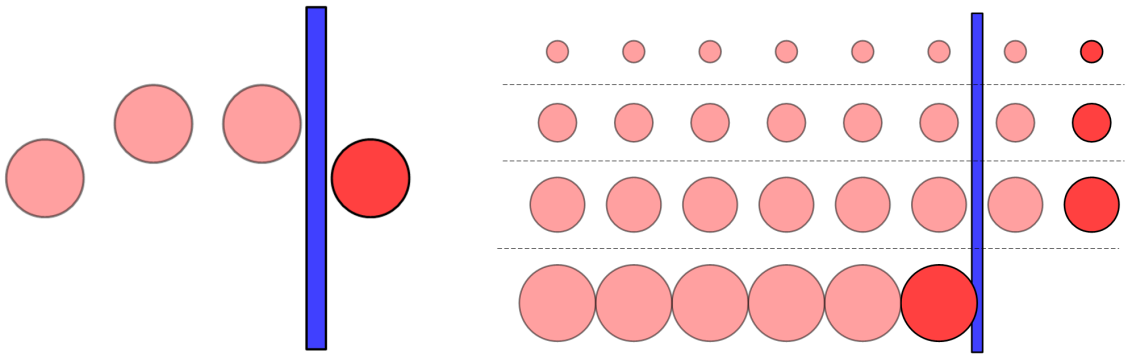


FIG. 8 – À gauche, la balle a traversé le mur en une itération. À droite, le schéma montre qu'une petite balle a plus de chances de traverser le mur.

Pour contourner ce problème, on peut soit augmenter le nombre d'itérations de calculs par seconde, soit implémenter un algorithme de collisions continues. On peut par exemple calculer une forme de balayage qui évitera la plupart des effets de tunneling en incluant peu de faux positifs. Avec cet algorithme, on pourra aussi prédire l'instant exact de la collision.

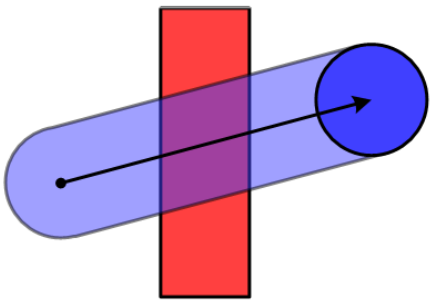


FIG. 9 – Un algorithme calcule une forme de balayage à partir de la balle, ce qui permet de prédire qu'elle va entrer en collision avec le mur.

Pour obtenir plus de précisions quant aux différentes méthodes de mouvement et de collision dans un moteur physique, veuillez vous référer à la présentation de *Squirrel Eiserloh*[6].

Dans la pratique, la détection de collisions continues de *Box2D* est activée par défaut pour les objets statiques, car elle demande peu de calculs supplémentaires dans ce cas précis. Mais on peut aussi l'activer pour un objet dynamique avec la fonction suivante :

```
object->SetBullet(true);
```

Conclusion

Au cours de ce travail, j'ai réussi à créer un jeu vidéo 2D en partant presque de zéro. L'utilisation d'un moteur physique a ajouté une grande complexité au projet, mais a grandement contribué à sa valeur. Je suis très satisfait du résultat.

Il est habituellement déconseillé de commencer la programmation par la création d'un jeu vidéo. Il est vrai que j'ai quelquefois été frustré par les défis techniques. Je ne regrette pourtant pas d'avoir outrepassé ce conseil. En effet, j'ai toujours réussi à surmonter les problèmes qui se sont posés à moi, moyennant parfois une bonne dose d'acharnement. En commençant tôt le travail et en avançant par petites étapes, j'ai pu remplir tous mes objectifs.

À l'avenir, si je devais commencer un nouveau projet, je fixerais peut-être des objectifs moins ambitieux, j'utiliserais un moteur de jeu complet et je chercherais un mentor qui pourrait répondre à mes questions plus rapidement que moi-même.

Après ce travail de maturité, je vais peut-être continuer ce projet ou participer à l'extension d'un projet existant.

Bibliographie

[1] Mike Banahan, *The C Book*, http://publications.gbdirect.co.uk/c_book/, 2003.

[2] Sean Chapel, *Complete 2D Engine Overview*, http://wiki.gamedev.net/index.php/SDL:Tutorials:Complete_2D_Engine_Overview, dernière modification 24 mai 2006.

[3] Wikibooks, *C++ Programming*, http://en.wikibooks.org/wiki/C++_Programming, dernière consultation novembre 2008.

[4] Chris Hecker, *Rigid Body Dynamics*, http://chrishecker.com/Rigid_Body_Dynamics, 2007.

[5] Erin catto, *Box2D Manual*, http://www.box2d.org/manual.html, 2008.

[6] Squirrel Eiserloh, *Motion and Collision - It's All Relative*, http://www.eiserloh.net/gdc2006_Eiserloh_Squirrel_PhysicsTutorial.ppt, 2006.

Outils et licence

Ce rapport a été écrit entièrement en LaTeX à l'aide de la police d'écriture Charter et des paquets *geometry*, *babel*, *ucs*, *inputenc*, *url*, *listings*, *fontenc*, *hyperref*, *graphicx*, *wrapfig*, *float* et *fancyhdr*.

La totalité de ce travail a été réalisée sous *Ubuntu* avec les outils *Gedit* pour l'édition du code et du rapport, *Inkscape* et *GIMP* pour l'édition des images. *Code : :Blocks* a été utilisé pour la création du binaire Windows.