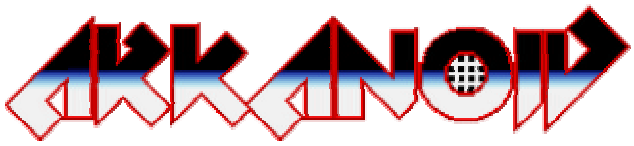


Choix et Motivation :

J'ai choisi de refaire une version d'Arkanoid par nostalgie des jeux d'arcade des années 80.
En effet, lors de sa sortie en salle, Arkanoid m'a tout de suite passionné, au point d'y passer toutes mes économies. J'ai fini ce jeu plusieurs fois en arcade, puis également sur la version Amiga.
Il n'existe malheureusement pas de version PC à la hauteur de la version arcade.
Je me suis donc décidé à relever le défit de remédier à ce manque et de faire ma version d'Arkanoid en Java (JDK 1.4.x Win32)

Historique :

Petit jeu dans son concept, Arkanoid est devenu l'un des chef-d'œuvres de Taito qui en 1986, en arcade [Main Processor: Z80 (8-bit; 6 MHz); M68705 (0.49 MHz)] remettait au goût du jour le genre du casse-brique, un principe on ne peut plus élémentaire : le joueur dirige une raquette (*une petite barre*) et doit détruire des murs composés de nombreuses briques au moyen d'une balle. Arkanoid sur Amiga fut adapté par Discovery Software International, et c'était à l'époque de sa sortie, la conversion parfaite du jeu d'arcade, ce qui en faisait alors la meilleure version disponible sur micro. On n'en attendait pas moins des auteurs des meilleurs titres de l'Amiga : *Martin Pedersen* et *Torben P. Larsen*, qui réalisèrent Hybris, Battlesquadron et Sword of Sodan.

Règles du jeu :

Briques normales :
On peut casser les briques normales simplement en les frappant une fois avec la balle. 50 à 120 points sont attribués selon la couleur.

Briques solides :
Il faut toucher ces briques plusieurs fois avec la balle pour les détruire.
Le nombre de fois requis est le suivant :

2 fois	1er au 8ème niveau
3 fois	9ème au 16ème niveau
4 fois	17ème au 24ème niveau
5 fois	25ème au 32ème niveau

Les points sont donnés lorsque la brique est cassée. 50 points*niveau

Murs indestructibles :
Ces briques ne peuvent pas être détruites.
Certaines briques contiennent des capsules bonus :

(S) Slow down	Ralentit la balle
(C) Catch & fire	Effet glue
(E) Expand	Augmente la taille de la raquette
(D) Divide	Triple balles
(L) Laser beam	La raquette est munie de laser
(B) Break	Permet d'aller au niveau suivant
(P) Player addition	Vie supplémentaire

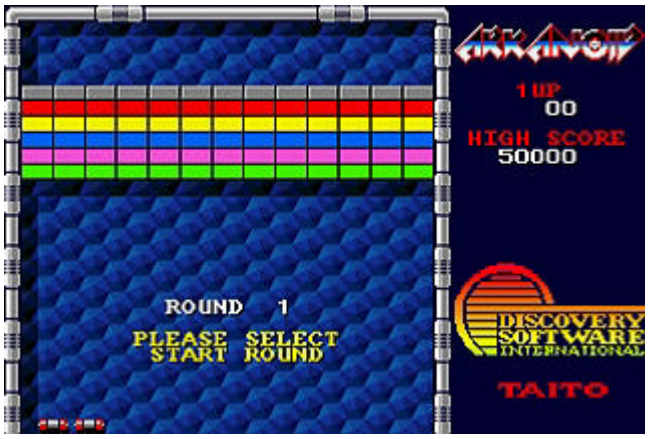
Les capsules bonus sont effectives jusqu’à ce que le joueur perde une vie, que le round soit terminé ou qu’une autre capsule soit prise.

1000 points sont donnés pour chaque capsule.

Les monstres apparaissent en haut de l’écran et passent par les briques cassées. Les détruire donne 100 points.

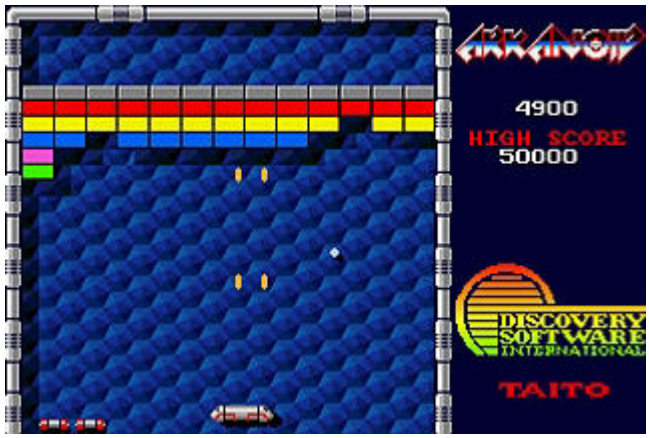
Le niveau est terminé lorsque toutes les briques ont été cassées.

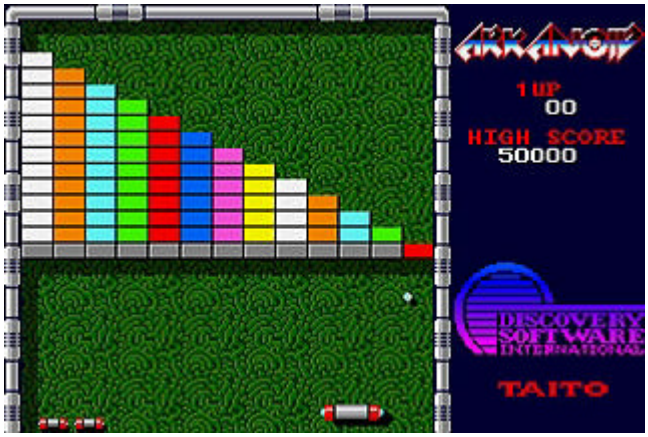
Présentation des niveaux :



Petite option très pratique et qui ne figure pas sur toutes les adaptations du jeu, l'écran de sélection des niveaux. Le programme nous permet, en effet, de commencer à n'importe lequel des 20 premiers niveaux. Au total il y en a 32, plus un pour DOH, le Boss de fin.

Une fois cassées, certaines briques laissent échapper des dalles (bonus) qui descendent vers la "raquette" du joueur. Il suffit de les réceptionner pour voir les propriétés de la "raquette" modifiées : allongement, magnétisme, triple-balles, vie supplémentaire, téléporteur vers le niveau suivant, etc. Ici, ma "raquette" dispose de deux canons lasers. Comme cela facilite la tâche de démolition, les programmeurs ont implémenté un algorithme qui accélère la vitesse de la balle : stressant !





Comme le principe est simple, et que l'intérêt du jeu ne repose pas sur un scénario, les auteurs ont dû se creuser pour proposer des niveaux captivants. Le "level-design" est l'un des points forts d’ Arkanoid. Les propriétés des briques sont différentes suivant leur couleur, ce qui autorise une multitude de possibilités, et de cheminements logiques pour la balle. Un exemple est illustré ici, avec le deuxième niveau : en cassant la brique rouge de droite, on ouvre une porte qui permet de faire entrer la balle dans la partie haute du "mur de briques". Avec un peu de chance, on peut se reposer quelques secondes…

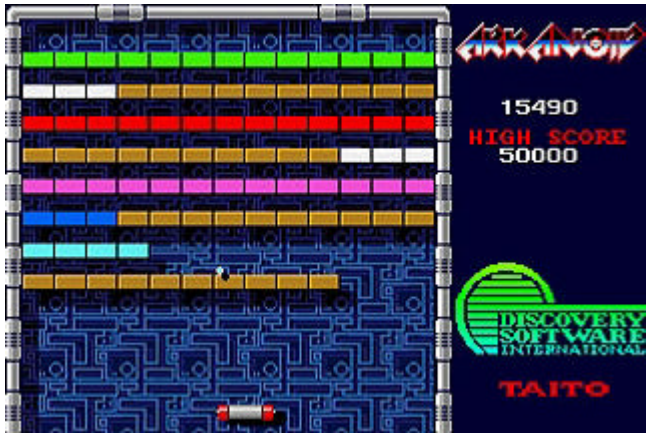
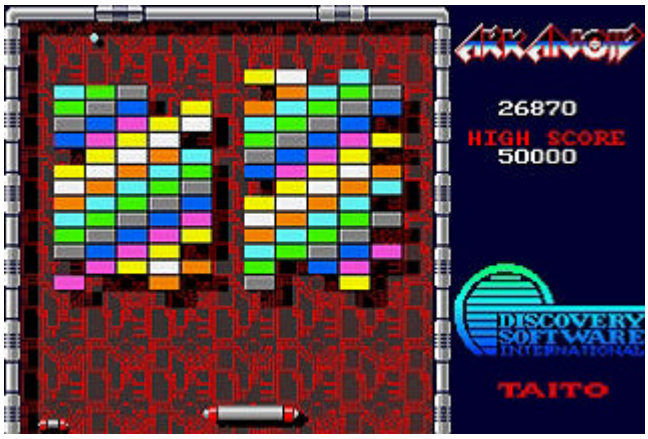
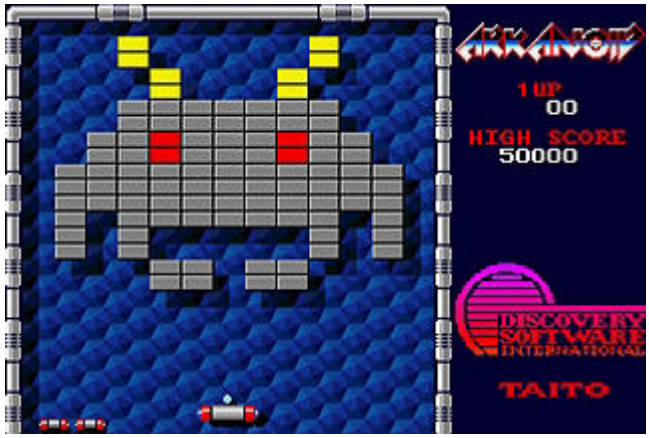


Illustration parfaite du bon level-design d'Arkanoid : les briques de couleurs sont enfermées dans un couloir en "S" formé par des briques incassables. Pour tout détruire, il faut trouver le meilleur "angle" pour jouer avec les rebonds. Là encore, avec un peu de savoir-faire, la balle fera le travail à votre place, et ne redescendra plus avant d'avoir tout cassé !



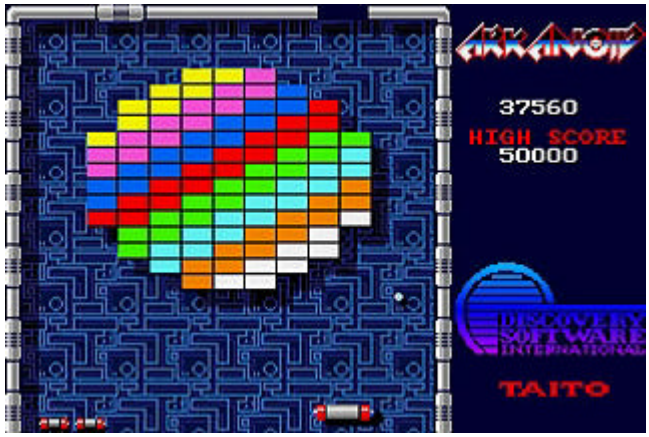
Au quatrième niveau, le joueur est confronté à une autre difficulté : les monstres évoluent en liberté, puisqu'il n'y a plus de mur de briques entre la "raquette" et les générateurs de monstres (*les deux cylindres sur la barre transversale*). Aux rebonds prévisibles contre les parois et les briques, les monstres – qui ne peuvent pas être plus de trois simultanément sur l'écran – provoquent des rebonds aléatoires…



Gros clin d'œil à Space Invader, grand classique de Taito : le cinquième tableau voit le joueur affronter l'un des "sprites" du jeu original grossi 25 fois !

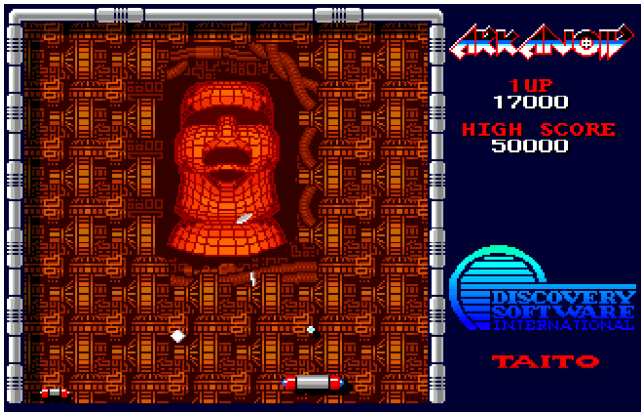
Le bonus triple-balles en action : la balle initiale se divise en trois parties égales qui se dirigent dans trois directions divergentes. Les suivre est assez difficile, même si curieusement, on parvient assez aisément à en garder deux, assez longtemps. Un niveau peut être terminé rapidement grâce à ce bonus, qui ne vaut cependant pas le laser !

Un niveau facile, très bariolé et acidulé qui annonce tout de suite la couleur : "ce sont les vacances" ! Comptez une minute avec trois balles, et dix secondes avec les lasers, on se croirait alors dans un shoot'em up… Mais profitez-en car les niveaux qui suivent seront plus retors.



La "dernière brique" ! Alors que l'on peut venir à bout de toutes les briques en quelques coups de raquettes bien sentis, la "dernière brique" peut nous faire tourner en bourrique et épuiser toutes nos vies. Pourquoi ? Simplement parce qu'il n'y a plus autant de rebonds possibles avec les autres briques, les trajectoires doivent donc être créées par le joueur à l'aide de sa "raquette". Une science qui ne s'avère pas exacte et qui se complique avec la présence des monstres.

DOH, le boss de fin (*tête d'une statue de l'Île de Pâques*). Il est difficile d’en venir à bout car il faut le toucher à maintes reprises avec la balle, et en plus il crache des tuiles qui suivent le mouvement de la raquette.



Implémentation :

Pour une question de temps, toutes les options présentées ci-dessus n'ont pas été implémentées.

Implémenté :
- Tous les niveaux originaux
- Graphisme refait selon modèle original
- Rebond aléatoire (balle et raquette)
- Effet d'ombre
- Logo avec raster arc-en-ciel
- Musique et sons optimisés et remis au goût du jour

Non implémenté :
- Bonus
- Monstres
- Sélection des niveaux
- Niveau final
- Explosion de la raquette

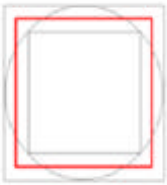
Détection de collision :

Il s'est vite avéré après quelques essais de faisabilité que la chose ne serait pas si facile.

S'il n'y a qu'une raquette et qu'une à trois balles, il y a en revanche 221 briques.

Mon premier algorithme de détection de collision fonctionnait sur un principe simple :

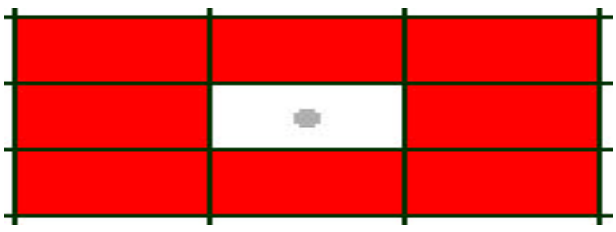
On définit un carré pour la balle, ce carré est situé à mi-chemin entre le carré qui se trouve à l'intérieur de la balle et celui qui se trouve à l'extérieur, puis un rectangle pour chaque brique.



Pour chaque image on calcule si la balle entre en collision avec une brique (s'il y a intersection entre le carré de la balle et le rectangle de la brique), et comme il y a 24 image/sec. et 221 briques, cela fait 5304 tests (appel à une méthode) par seconde et par balle. Il faut encore compter que dans la fonction de détection de collision se trouvent plusieurs tests.

Il faut tout d'abord cherché à diminuer le nombre de tests, pour ceci on peut avoir recours à la méthode suivante :

Comme on connaît les dimensions du plan de jeu, des briques ainsi que leur taille, il est facile de définir une grille virtuelle dans laquelle se trouvent les briques. Pour chaque image on calcule dans quelle cellule se trouve la balle et on ne fait le test de collision que pour les briques qui se trouvent autour de cette cellule.



On passe donc de 5304 tests à seulement 192 par seconde, c'est déjà un gros gain, mais ce n'est pas suffisant.

Il faut donc trouver un autre algorithme de détection de collision qui me permette à la fois d'avoir un minimum de tests à effectuer et qui donnera un résultat correct.

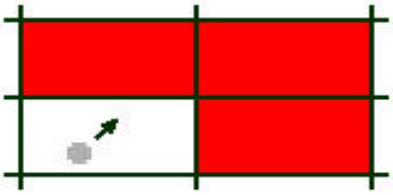
On conserve l'idée de la grille virtuelle, mais on utilisé la direction de la balle comme indicateur, car selon cette direction, la balle ne peut toucher que certaines briques.

Explications :

Prenons le cas où la balle bouge de gauche à droite et de haut en bas (cf. flèche)

Une fois déterminée la cellule dans laquelle se trouve la balle, nous avons les trois cellules qui pourraient être touchées à la prochaine itération.
Plusieurs cas de figure peuvent se présenter :

A) Les trois cellules contiennent des briques :



B) Deux cellules contiennent des briques :

C) Une cellule contient une brique :

D) Aucune brique

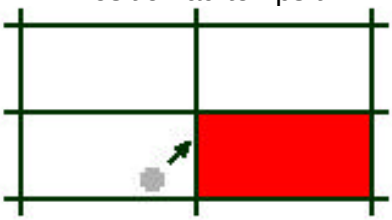
Quel que soit le cas, trois tests de collision suffisent, on passe alors de 192 tests à 72 tests.

Malgré ceci, il y a toujours des problèmes :

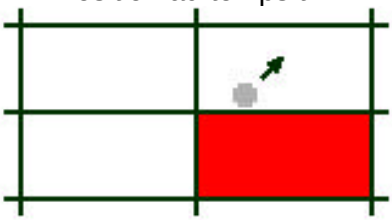
- La balle peut toucher une brique ailleurs que sur un bord
- La balle se bloque
- La balle se glisse entre les briques
- La balle casse la brique du coin alors qu'il y a des briques de côté
- Etc…

Les problèmes viennent du fait qu'à chaque itération on calcule la position de la balle à l'image suivante pour déterminer la collision à l'avance, ce qui peut amener au cas suivant :

Position au temps t



Position au temps t+1



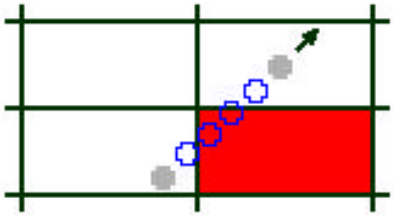
On remarque tout de suite que si la vitesse de la balle est trop grande, il se peut qu'elle passe à travers une brique, ce qui n'est évidemment pas souhaitable.

Pour obtenir une détection correcte il a fallu utiliser beaucoup d'approches différentes, utilisation d'un vecteur entre la position au temps t et t+1, avec un résultat assez moyen. Ensuite à l'aide d'un polygone sans plus de succès.

Grâce à la solution utilisée ci-dessus (utilisation de la direction de la balle comme indicateur) on a la possibilité de connaître quel côté de la brique a été touché.

Par exemple, dans le premier cas, la balle ne peut toucher que le bas de la brique du dessus, et le côté gauche de la brique de droite.

En gardant la solution du carré autour de la balle, mais cette fois-ci au lieu de calculer la position de la balle à l'image suivante, on fait abstraction de 'la réalité visible', et on calculé de manière beaucoup plus précise à l'aide de nombres flottants toutes les itérations entre une position et la suivante.



Dès que l'algorithme détecte une collision, il envoie l'image à l'écran, ainsi on a vraiment l'impression que la balle touche la brique.

A également été ajouté :

- la notion de priorité, car selon la direction de la balle, il y a plus ou moins de chance qu'elle touche une brique plutôt qu'une autre, ceci est dû au fait que les briques ne sont pas carrées.

- Un petit ajout aléatoire dans le rebond pour éviter toute possibilité de blocages de la balle.

Ce qui donne l'algorithme de détection suivant : (cf. schéma de l'algorithme en annexe)

```
for(int iter = 0; iter < ballSpeed; iter++) {

        //Calculate next ball position
        ballPosX -= ballDX;
        ballPosY -= ballDY;

        //Calculate ballRect position
        ballRect.x = ballPosX;
        ballRect.y = ballPosY;
```

```
		//Calculate ball position in game area
		cY = (int)Math.floor((ballPosY+ballHalfHeight-BORDER_SIZE) / BRICK_HEIGHT);
		cX = (int)Math.min(Math.floor((ballPosX+ballHalfWidth- BORDER_SIZE) / BRICK_WIDTH), BRICK_X-
1);


//Test only if ball is in brick area
if(cY <BRICK_Y) {

//Test if ball move down
    if (ballDY < 0) {

		//Test if ball move right
		if(ballDX < 0) {
			if(testCollision(BOTTOM)) {
				ballDY = -((Math.random()/40+1)*ballDY);
				if(testCollision(RIGHT)) {
					ballDX = -((Math.random()/40+1)*ballDX);
				}
				return;
			}
			else if(testCollision(RIGHT)) {
				ballDX = -((Math.random()/40+1)*ballDX);
				return;
			}
			else if(testCollision(BOTTOM_RIGHT)) {
				ballDY = -((Math.random()/10+1)*ballDY);
				ballDX = -((Math.random()/10+1)*ballDX);
				return;
			}
		}
		//Test if ball move left
		else {
			if(testCollision(BOTTOM)) {
				ballDY = -((Math.random()/40+1)*ballDY);
				if(testCollision(LEFT)) {
					ballDX = -((Math.random()/40+1)*ballDX);
				}
				return;
			}
			else if(testCollision(LEFT)) {
				ballDX = -((Math.random()/40+1)*ballDX);
				return;
			}
			else if(testCollision(BOTTOM_LEFT)) {
				ballDY = -((Math.random()/10+1)*ballDY);
				ballDX = -((Math.random()/10+1)*ballDX);
				return;
			}
		}
 }
 else {    //Test if ball move up
		if(ballDX < 0) {  //Test if ball move right
			if(testCollision(TOP)) {
				ballDY = -((Math.random()/40+1)*ballDY);
				if(testCollision(RIGHT)) {
					ballDX = -((Math.random()/40+1)*ballDX);
				}
				return;
			}
			else if(testCollision(RIGHT)) {
```

```
                        ballDX = -((Math.random()/40+1)*ballDX);
                        return;
                }
                else if(testCollision(TOP_RIGHT)) {
                        ballDY = -((Math.random()/10+1)*ballDY);
                        ballDX = -((Math.random()/10+1)*ballDX);
                        return;
                }
        }
        else {    //Test if ball move left
                if(testCollision(TOP)) {
                        ballDY = -((Math.random()/40+1)*ballDY);
                        if(testCollision(LEFT)) {
                                ballDX = -((Math.random()/40+1)*ballDX);
                        }
                        return;
                }
                else if(testCollision(LEFT)) {
                        ballDX = -((Math.random()/40+1)*ballDX);
                        return;

                }
                else if(testCollision(TOP_LEFT)) {
                        ballDY = -((Math.random()/10+1)*ballDY);
                        ballDX = -((Math.random()/10+1)*ballDX);
                        return;
                }
        }
 }
}//If in brick area
```

Et voici la méthode testCollision() :

```
private final boolean testCollision(int pPos) {

    hitted = false;

    switch(pPos) {

        case TOP :
            cX2 = cX;
            cY2 = Math.max(cY-1,0);
            break;

        case TOP_RIGHT :
            cX2 = Math.min(cX+1,LevelsLoader.BRICK_X-1);
            cY2 = Math.max(cY-1,0);
            break;

        case RIGHT :
            cX2 = Math.min(cX+1,LevelsLoader.BRICK_X-1);
            cY2 = cY;
            break;

        case BOTTOM_RIGHT :
            cX2 = Math.min(cX+1,LevelsLoader.BRICK_X-1);
            cY2 = Math.min(cY+1,LevelsLoader.BRICK_Y-1);
            break;

        case BOTTOM :
            cX2 = cX;
            cY2 = Math.min(cY+1,LevelsLoader.BRICK_Y-1);
```

```
                break;

        case BOTTOM_LEFT :
                cX2 = Math.max(cX-1,0);
                cY2 = Math.min(cY+1,LevelsLoader.BRICK_Y-1);
                break;

        case LEFT :
                cX2 = Math.max(cX-1,0);
                cY2 = cY;
                break;

        case TOP_LEFT :
                cX2 = Math.max(cX-1,0);
                cY2 = Math.max(cY-1,0);
                break;

    }

    if(level[cY2][cX2][LevelsLoader.EXIST] == 1) {
        brickRect.x = level[cY2][cX2][LevelsLoader.POS_X]+1;
        brickRect.y = level[cY2][cX2][LevelsLoader.POS_Y]+1;

        if (brickRect.intersects(ballRect)) {

                playSound(2, false);
                hittedBricks.add(new Point(cY2, cX2));
                if(!demo) {
                    score += level[cY2][cX2][LevelsLoader.POINTS];
                    winLife += level[cY2][cX2][LevelsLoader.POINTS];
                    if(winLife >= WIN_LIFE) {
                                winLife = 0;
                                lives++;
                                drawLives();
                    }
                }
                if(level[cY2][cX2][LevelsLoader.TYPE]<9) {
                    level[cY2][cX2][LevelsLoader.STRENGTH]--;
                }
                if(level[cY2][cX2][LevelsLoader.TYPE]>7) {
                    level[cY2][cX2][LevelsLoader.IMAGE] = 7;
                }
                                hitted = true;
        }
    }

    return hitted;
}
```

On arrive avec cet algorithme à déterminer le moment exact de l'impact de la balle sur la brique, et donc à la faire rebondir de la bonne manière au bon moment ce qui élimine tous les effets de bords.

Il est clair que le nombre de calculs est important, mais ce sont des calculs très simples pour un processeur et l'impact sur la performance est négligeable.

Optimisation :

Pour gagner en efficacité, il faut quitter le modèle objet et coder de manière plus procédurale à l'aide de tableaux de primitives à la place d'objets. A ce point du développement, il est nécessaire de reprendre à zéro le code afin de bénéficier de l'expérience acquise avec le modèle objet.

Le travail s'avère payant, il y a un gain net de vitesse, mais ce n'est toujours pas suffisant pour être jouable.

Je cherche donc où je pourrais encore améliorer la vitesse.
En partant du fait que l'on dois avoir un algorithme de détection assez efficace, il faut investiguer dans une autre direction : l'affichage graphique

On base les recherches sur les points suivants :

- Double buffering

Le double buffering évite l'effet de scintillement que l'on observe lors de l'animation simplement en calculant la prochaine image à afficher en arrière plan, on ne voit donc pas les objets se dessiner les uns après les autres à l'écran, mais une image complète apparaît à l'écran.

Pas de gain de performance, mais une meilleure impression de fluidité.

- Diminution de la zone à redessiner

En diminuant la zone à redessiner à l'écran, on réduit le besoin en bande passante entre la mémoire et la carte graphique, car on ne modifie qu'une petite partie de l'image à la fois.
Pour une image d'une résolution de 640x480x16, il faut 600Kb par image, donc ~14Mb par seconde de bande passante. C'est beaucoup trop pour le petit processeur graphique du portable.

Si le principe paraît simple, la réalisation ne l'est pas.

Il faut en effet tenir compte que la surface de jeu est composée de plusieurs couches :
- Fond d'écran (texture)
- Couche d'ombres
- Couche avec balle, raquette et briques

Il est donc nécessaire d'utiliser plusieurs images en mémoire (3) afin d'être à même de redessiner ce qui se trouvera derrière la balle, la raquette et les briques lorsque ceux-ci bougent.

Au final, le gain est vraiment impressionnant que ce soit sur un portable ou sur un PC 2Ghz.

- Méthodes d'affichage plus efficaces

En Java, on utilise les méthodes update() et paint() pour mettre le code qui effectuera l'affichage et la méthode repaint() pour indiquer que l'on désire rafraîchir cette image.
Le problème, c'est que l'on ne maîtrise pas quand cette méthode va réellement être appelée.

Pour palier à ce problème, il « suffit de » ré implémenter le processus d'affichage (toutes les méthodes en une seule), comme ceci on maîtrise parfaitement quand la mise à jour est effectuée, de plus on évite des appels à des méthodes, donc encore un petit gain.

- Images en cache mémoire

Les cartes graphiques sont conçues pour manipuler des images de manière plus rapide si elles sont stockées dans leur mémoire. On dispose depuis Java 1.4, d'un nouveau type d'image, les VolatileImage.

Cet objet a la particularité de travailler avec une image stockée dans la mémoire cache de la carte graphique, on obtient de cette manière un gain en vitesse de traitement de l'ordre de 3x.

Il faut juste faire attention au fait que les anciennes cartes graphiques ne disposaient pas d'une mémoire de 64Mb comme les cartes actuelles, dans le cas du portable, on dispose uniquement de 4Mb.

Une image en 640x480x16 occupe 600Kb, on utilise donc 1800 Kb pour les trois images, ce qui permet de mettre toutes les couches en mémoire.

La vitesse est maintenant impressionnante, au point même que le jeu est injouable sur un portable et un PC 2Ghz.

Avec toutes ces optimisations, le gain doit avoisiner les 20x par rapport à la première version.

Contrôle de la vitesse d'animation :

Afin de palier maintenant à ce surplus de vitesse, il faut trouver un moyen d'avoir une vitesse de jeu identique à la fois sur le portable et sur le PC 2Ghz.

Que ce soit pour le cinéma, la télévision ou les jeux vidéo, on utilise une suite d'images fixes qui défile à une vitesse constante pour donner l'impression d'animation.
Ceci fonctionne grâce à la persistance de la vision, en fait l'œil humain retient l'image environ 1/20 de seconde après l'avoir vue.
Le zeotrope inventé par William George Horner en 1834 est constitué d'une série d'image sur une bande de papier et fonctionne sur ce principe.
On sait que pour avoir un effet d'animation, il faut un minimum de 24 images/sec. afin que l'œil ne distingue pas le passage d'une image à l'autre, ce principe est toujours utilisé de nos jours pour le cinéma.


Le zeotrope


une pellicule 35mm.

Pour une seconde à 24 images/sec, une image a une durée d'affichage de 1000/24 = ~42 ms.

Il faut donc trouver un moyen quelle que soit la vitesse de l'ordinateur d'obtenir cette vitesse de rafraîchissement.

La solution se trouve au niveau du thread d'animation dont voici le code simplifié :

```
private static int FRAME_DURATION = 42;

while (Thread.currentThread() == animation) {
        t = System.currentTimeMillis() + FRAME_DURATION;

        //Ici on effectue les calculs et l'affichage

        try {

                sleepTime = t - System.currentTimeMillis();

                if(sleepTime <= 0)    {
                        Thread.yield();
                }
                else {
                        Thread.sleep(sleepTime);
                }

         }
        catch (InterruptedException e) {
                break;
        }
}
```

Si l'on analyse ce code, on remarque que l'on prend le temps avant d'effectuer les calculs et l'affichage, puis le temps après.

En calculant la durée que l'ordinateur a mise pour effectuer cette opération, il suffit simplement de faire 'dormir' le thread d'animation de la durée qui manque pour obtenir les 42 ms

Exemple :

Pour viser une animation à 24 images/sec. on fixe la constante FRAME_DURATION à 42.

On prend le temps t : 0 (On prend 0 pour l'exemple, dans la réalité, c'est le temps en milliseconde qui est retourné) et on lui ajoute la valeur de la constante FRAME_DURATION → t = 0 + 42 = 42.

On effectue les calculs et l'affichage : durée = 32ms

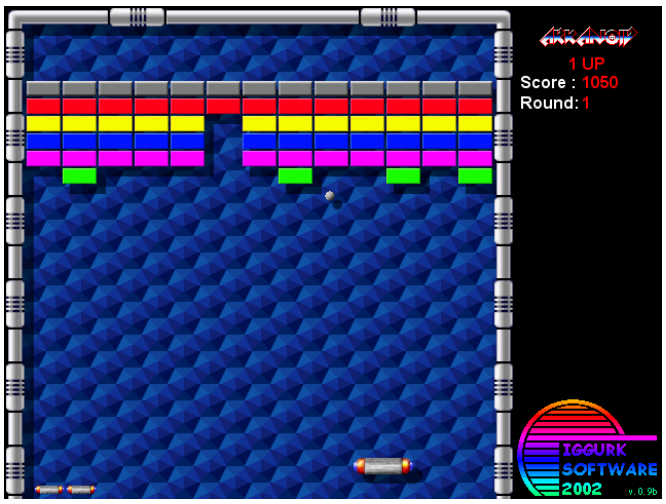
On calcule la différence de temps par rapport à avant : 42ms – 32ms = 10ms

Comme la différence est plus grande que 0, on fait dormir le thread de cette différence, au total le temps de traitement d'une itération aura bien pris 42ms.

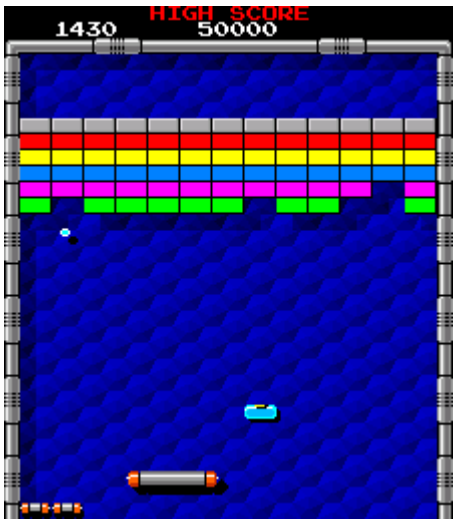
Notez que si la différence est négative nous n'aurons pas les 24 img./sec., ce qui donnera une animation saccadée.

Versions :

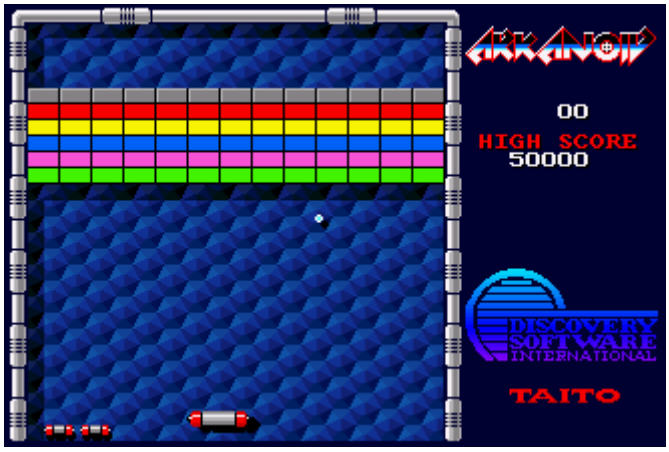
Ma version :



Arcade:

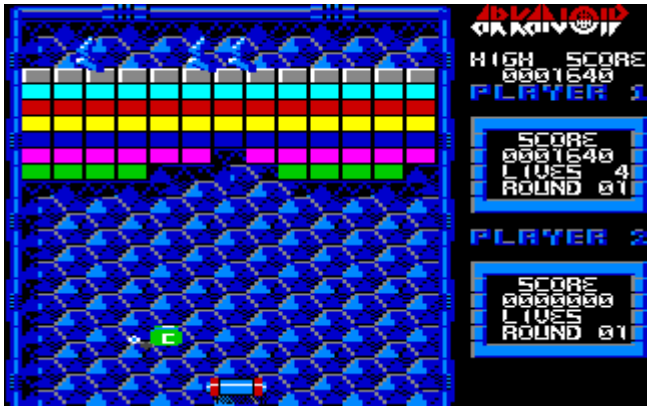


Amiga :

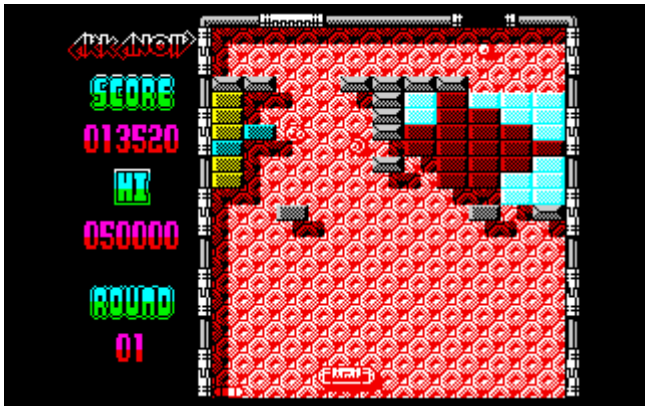


PC Dos :

C64 :



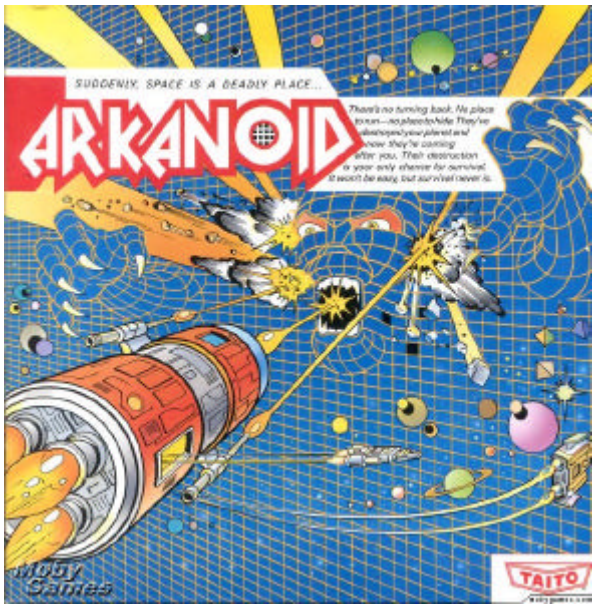
Amstrad CPC :



Images :


ARKANOID
TAITO
DISCOVERY SOFTWARE INTERNATIONAL


ARKANOID


SUDDENLY, SPACE IS A DEADLY PLACE...
ARKANOID
TAITO