Projet C++

CSII 1

Beau Morgan
Courazier Nicolas

SOMMAIRE

Introduction

Avant toute chose, nous allons brièvement rappeler le sujet de ce projet. Ceci est important si un autre équipe de développement reprend le projet afin d'y apporter quelques modifications.

Le projet est la simulation d'un aquarium où vivent d'étranges animaux à savoir les **Speedbricks** et les **Cutits**. Ce sont des organismes vivant qui **vivent**, **grandissent**, **s'accouplent**, se **coupent** et **meurent**.

Les Cutits sont des bactéries qui vivent dans notre aquarium. Ces bactéries se déplacent en groupes et de façon linéaire horizontalement. La particularité de ces bactéries est qu'elles peuvent « découper » (ou plutôt scinder) un Speedbrick en deux. En effet, lors d'une collision entre un Cutit et un Speedbrick, celui-ci est coupé et apparait une autre tête sur le morceau coupé.

Les Speedbricks sont des petits animaux constitué d'une tête d'une couleur qui dépend des ses parents, nous en reparlerons plus tard, d'un corps d'une multitude d'anneaux correspondant au nombre d'années vécues et d'un sexe. En effet, un Speedbrick possède un âge, et à chaque anniversaire il se voit rajouter un anneau à son corps. Lorsque son heure a sonné (que son âge arrive au maximum), alors celui-ci meurt. Le sexe d'un animal est facilement détectable, une couleur est attribuée pour les mâles et pour les femelles.

En addition à cela, les Speedbricks peuvent se reproduire entre eux du moment que le sexe est opposé. Cependant, une gestion de code génétique est implémentée, la consanguinité à des conséquences sur les Speedbricks à tout point de vue. Un compteur génétique est implémenté et selon la configuration de l'application présente des mouvements différents afin de limiter leur possibilité de reproduction (simule un handicap). La couleur de la tête dépend de celle de ses parents, celle-ci prend les couleurs des deux parents et ainsi de suite.

Ce projet à été développé selon différentes phases. Comme pour tout projet nous avons commencé par une phase d'analyse préliminaire et détaillée via la méthode **UML** pour faire état de tous les problèmes.

Plusieurs paramètres sont à prendre en compte car cette application comporte de nombreuses subtilités. La couche application fut développée en C++ en utilisant l'API (Application Programming Interface) graphique **OpenGL**. Ensuite, afin de pouvoir utiliser notre simulation d'aquarium sur les plateformes Windows et UNIX nous avons respecté la norme **ANSI** (American National Standards Institute) et utilisé les fonctions standards de la **STL** (Standard Template Library).

Afin d'obtenir une application modulaire et facilement maintenable, nous avons veillé à dissocier la couche **métier** de la couche **présentation**. En addition à cela, nous avons appliqué le modèle **PAC** (Présentation Abstraction Contrôle) permettant une répartition simple et efficace de la charge de travail au sein de l'équipe de développement.

Cette répartition permet à chaque participant au projet d'effectuer toutes les actions possibles et surtout valider les tests unitaires de l'application.

Vous trouverez dans ce document l'analyse du projet ainsi que les principaux algorithmes ayant servi à résoudre les principaux problèmes de l'application.

Classes principales

Les organismes

Il est ici important de bien dissocier les acteurs du système. Nous avons la chance d'avoir ici un cas simple, ne comportant qu'une population de deux types d'animaux vivant dans un espace clos, à savoir leur aquarium.

Les deux espèces peuvent se déplacer dans leur espace, et dispose donc des mêmes caractéristiques de déplacement :

- Vélocité (vecteur de déplacement sur X, Y et Z)
- Emplacement (coordonnées en X, Y et Z)

Les Speedbricks possèdent des caractéristiques définies :

- Age Actuel
- Corps (constitué d'anneaux)
- Sexe (mâle, femelle ou indéterminé pour les plus jeunes)
- Une carte génétique (pour la consanguinité)

Ces attributs sont les principaux, viendrons s'y ajouter d'autres pour la gestion du temps, de l'insensibilité face aux collisions, etc…

Quels sont les possibilités offertes aux Cutits ?

Aucune spécificité n'est réellement incluse dans les Cutit. Ces organismes permettent de couper les Speedbricks, mais ne dispose d'aucune action sur eux-même et/ou ses congénères.

Quels sont les possibilités des Speedbricks ?

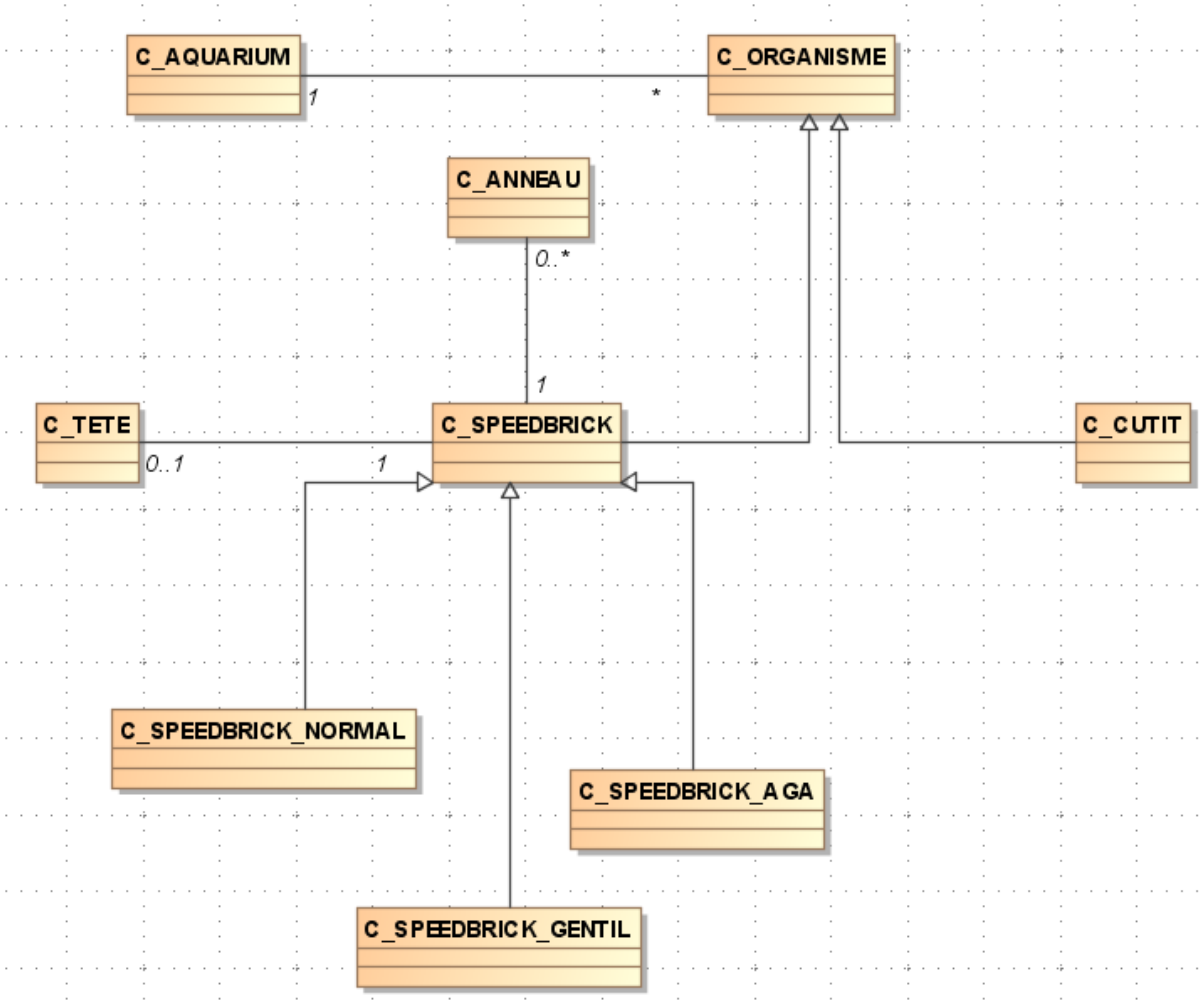
Ce dernier est capable **grandir**, de se **reproduire**, et de se **couper** lors d'une collision avec un Cutit. En ce qui concerne l'accouplement il est évident que les deux Speedbricks doivent être de sexe différent.

Quels sont les possibilités communes à tous les animaux ?

Ils sont capables de se **déplacer** et de **s'afficher** sur l'écran.

Le fait que les deux animaux se déplacent de la même façon va nous permettre de mettre en place une classe abstraite représentant un animal. Ensuite, nous auront deux classes filles qui seront Speedbrick et Cutit qui hériteront des comportements en commun inclus dans la classe animal et les fonctions virtuelles.

Le diagramme de classe (très) simplifié est le suivant.



Afin de gérer cette simulation, nous avons mis en place une classe maitresse ayant connaissance de tout ce qui se déroule : le classe C_AQUARIUM. Celle-ci est donc un singleton. Elle possède une méthode permettant à toutes les autres classes de récupérer son instance.

L'unité de temps

Dans l'univers de nos amis Speedbrick et Cutit, le temps s'écoule de la même manière que sur Terre. Chaque jour, les Speedbrick vieillissent un peu plus et se rapprochent de leur mort. Chaque Speedbrick a une espérance de vie chiffrée en année. De plus, chaque année les Speedbrick grandissent et se voient doté d'un anneau supplémentaire.

Leur âge est exprimé en jour. Chaque jour représente un nouveau passage dans la boucle d'affichage du moteur 3D. Ainsi, une fréquence d'affichage de 50 images par secondes, chaque organisme vieilli de 50 jours par seconde. Cette fréquence est paramétrable dans le moteur graphique.

De plus, la vie des Speedbrick est rythmée par leurs anniversaires. En effet, une année est composée d'un certain nombre de jour. Une année est le temps nécessaire à la planète de nos organismes pour effectuer une rotation autour de leur soleil. Celle-ci est définie par une unité de temps. L'unité de temps est donc le nombre de jour formant une année.

De cette manière, un Speedbrick meurt si l'équation suivante est respectée :

$$\text{Durée de vie} \leq \frac{\text{age du Speedbrick}}{\textit{unité de temps}}$$

De la même manière, un Speedbrick fête son anniversaire si :

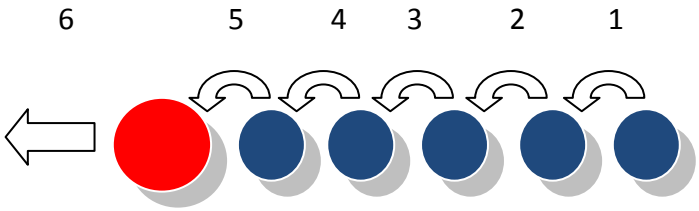
$$\text{age du Speedbrick \% unité de temps} == 0$$

C'est-à-dire si le résultat de la division entre l'âge du Speedbrick et l'unité de temps présente un reste nul, ou encore si l'unité de temps est un multiple de l'âge.

Beau Morgan
Courazier Nicolas

Les déplacements

Dans un premier temps, afin de déplacer les Speedbrick, nous avons effectué un déplacement itératif. Nous partions du dernier anneau, si le Speedbrick en possède, puis nous remontions jusqu'à la tête. Ainsi, chaque anneau n-1 prenait la place de l'anneau n. Le premier anneau prenait la place de la tête du Speedbrick puis celle-ci se déplaçait.

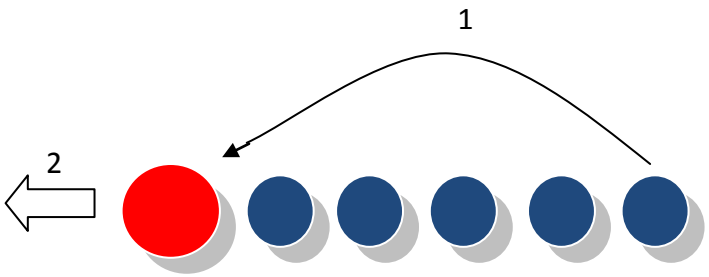


Exemple des Speedbrick normaux :

```
void C_SPEEDBRICK_NORMAL::seDeplacer(){
	int L_taille;
	L_taille = this->listeAnneaux.size();
	//si il y a des anneaux
	if (L_taille > 0) {
		for(int compteurAnneau = L_taille; compteurAnneau > 1; compteurAnneau--) {
			C_ANNEAU* anneauEnCours;
			anneauEnCours = this->listeAnneaux.at(compteurAnneau-1);
			C_ANNEAU* L_anneauPrecedent;
			L_anneauPrecedent = this->listeAnneaux[compteurAnneau-2];

			float L_posX;
			float L_posY;
			float L_posZ;
			L_posX = L_anneauPrecedent->getPosX();
			L_posY = L_anneauPrecedent->getPosY();
			L_posZ = L_anneauPrecedent->getPosZ();
			anneauEnCours->setPosition(L_posX, L_posY, L_posZ);
		}
		this->listeAnneaux[0]->setPosition(this->getPosX(),this->getPosY(),this-
>getPosZ());
	}

	this->coord.pos[0] = this->coord.pos[0] + this->vel.velo[0];
	this->coord.pos[1] = this->coord.pos[1] + this->vel.velo[1];
	this->coord.pos[2] = this->coord.pos[2] + this->vel.velo[2];

	if(this->listeAnneaux.size() > 0){
		this->onduler();
	}
}
```

Pour un Speedbrick de n anneaux nous avions dont n+1 (les n anneaux et la tête) itérations.

Puis nous avons décidé de modifier ce fonctionnement en intégrant un buffer circulaire. Pour chaque déplacement, ce procédé nous fait économise n-1 déplacements. En effet, seul le dernier anneau se déplace pour prendra la place de la tête puis la tête se déplace à son tour. Le dernier anneau devient le premier et l'avant dernier, le dernier.



Exemple des Speedbrick normaux après modification :

```
void C_SPEEDBRICK_NORMAL::seDeplacer(){
	this->deplacerAnneaux();

	this->coord.pos[0] = this->coord.pos[0] + this->vel.velo[0];
	this->coord.pos[1] = this->coord.pos[1] + this->vel.velo[1];
	this->coord.pos[2] = this->coord.pos[2] + this->vel.velo[2];

	if(this->listeAnneaux.size() > 0){
		this->onduler();
	}

	this->rebondMur();
	this->corrigerPosition();
}
```

Afin de faire onduler les Speedbricks normaux, nous avons recours à la fonction sinus. Celle-ci prend en paramètre le temps, c'est-à-dire le temps écoulé depuis le lancement de la simulation.

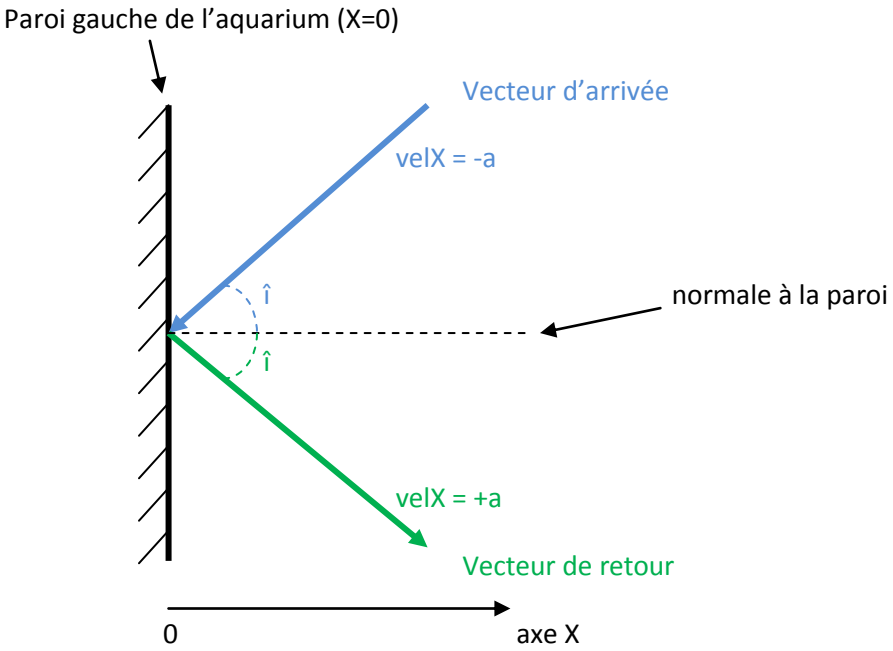
```
void C_SPEEDBRICK_NORMAL::onduler(){
	float L_sommeVecteurs;
	L_sommeVecteurs = abs(vel.velo[0]) + abs(vel.velo[1]) + abs(vel.velo[2]);
	float L_ratioX = (abs(vel.velo[1]) + abs(vel.velo[2])) / L_sommeVecteurs;
	float L_ratioY = (abs(vel.velo[0]) + abs(vel.velo[2])) / L_sommeVecteurs;
	float L_ratioZ = (abs(vel.velo[0]) + abs(vel.velo[1])) / L_sommeVecteurs;

	C_AQUARIUM* L_leAquarium;
	L_leAquarium = C_AQUARIUM::getAquarium();
	float L_ondulation;
	L_ondulation = sin((L_leAquarium->tempsEcoule)*0.5);

	this->coord.pos[0] = this->coord.pos[0] + (L_ratioX * L_ondulation);
	this->coord.pos[1] = this->coord.pos[1] + (L_ratioY * L_ondulation);
	this->coord.pos[2] = this->coord.pos[2] + (L_ratioZ * L_ondulation);
}
```

Les Cutit se déplacent en ligne droite, de manière horizontale. Il n'y a donc pas de déplacement sur l'axe Y. Lors de l'initialisation de la simulation, des bancs sont créés à des coordonnées et avec des vecteurs de déplacements aléatoire. Dans chacun de ces bancs se trouve un nombre donné de Cutit qui possèdent tous le même vecteur de déplacement. Ce mécanisme assure une cohésion du banc de Cutit tout au long de la simulation.

```
// Ajout des Cutits dans l'aquarium
for(int L_cptBanc = 0; L_cptBanc < P_nbBancCutit; L_cptBanc++){
	float L_centreBancX;
	float L_centreBancY;
	float L_centreBancZ;
	L_centreBancX = (rand()%this->getTailleX()*0.7) + 70;
	L_centreBancY = (rand()%(this->getTailleZ()-200) + 100);
	L_centreBancZ = (rand()%this->getTailleZ()*0.7) + 70;
	float L_velX;
	float L_velY;
	float L_velZ;
	L_velX = ((rand()%4)+1) - 2.5;
	L_velY = 0;
	L_velZ = ((rand()%4)+1) - 2.5;
	for(int L_cptCutit = 0; L_cptCutit < this->nbCutitParBanc; L_cptCutit++){
		C_CUTIT* L_unCutit;
		L_unCutit = new C_CUTIT;
		float positionX;
		float positionY;
		float positionZ;
		positionX = L_centreBancX + rand()%60;
		positionY = L_centreBancY + rand()%60;
		positionZ = L_centreBancZ + rand()%60;
		L_unCutit->setVelocite(L_velX, L_velY, L_velZ);
		L_unCutit->setPosition(positionX, positionY, positionZ);
		L_unCutit->ajouterOrganisme();
	}
}
```

Les virages

Un virage est déclenché à chaque approche d'une paroi de l'aquarium.
Un virage consiste à inverser le vecteur de déplacement normal à la paroi rencontrée.
Par exemple, lors de la « collision » entre un organisme et la paroi se trouvant à la limite X=0 de l'aquarium, le vecteur de déplacement en X (noté velX) est inversé.



Ainsi, on obtient un vecteur retour dont l'angle formé avec la normal à la paroi rencontrée à l'arrivée est identique à celui formé au retour. On a donc l'impression d'un rebond de l'organisme sur la paroi.

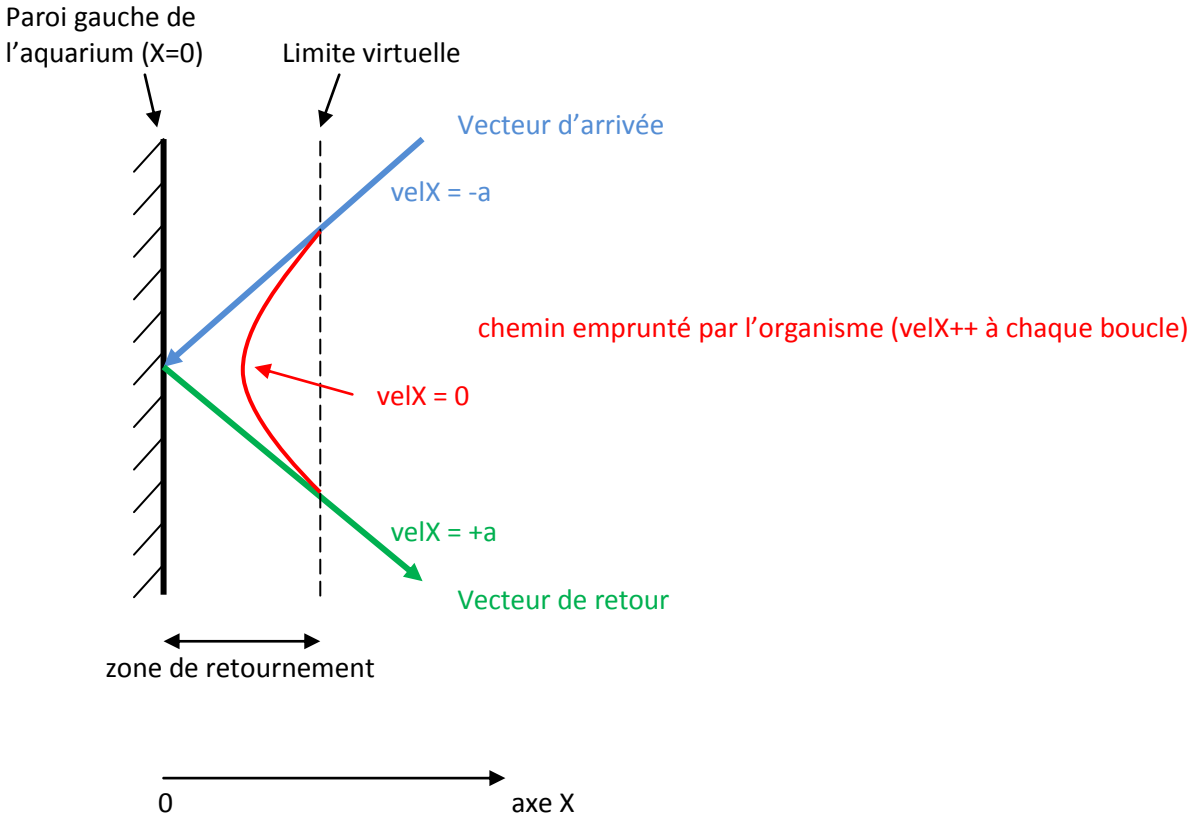
Ce principe offrant un rendu trop mécanique aux virages effectués par les organismes, il a été nécessaire de rendre ces derniers plus naturels.

Le retournement gracieux

Le retournement gracieux reprend le même principe que le rebond évoqué ci-dessus mais permet d'assouplir le changement de direction.

Pour cela, il est nécessaire de se rendre compte de la future collision bien avant que celle-ci ait lieue. Chacune des parois est donc « précédée » d'une limite invisible à partir de laquelle chaque organisme doit effectuer un retournement.

On peut donc ainsi rendre l'inversion du vecteur normal à la paroi progressive en l'incrémentant ou le décrémentant jusqu'à ce qu'on obtienne le vecteur de retour inverse.



Reproduction

Les Speedbricks tentent de faire perdurer leur espèce dès qu'ils en ont l'occasion. Dès qu'un mâle croise la route d'une femelle (ou inversement, les Speedbricks ne sont pas sexistes) en état de procréer, le processus de reproduction est lancé.

Les codes génétiques des deux parents sont fusionnés afin de former celui du fils. Si le code ainsi formé présente de trop grandes disparités, le fils présente des tarres.

```
void C_SPEEDBRICK::seReproduireAvec(C_SPEEDBRICK* P_unSpeedbrick) {
	bool reprodPossible1;
	bool reprodPossible2;
	reprodPossible1 = this->mettreEnReproduction();
	reprodPossible2 = P_unSpeedbrick->mettreEnReproduction();

	//le mâle est forcément false, seule la femelle peut être true
	if(reprodPossible1 == true || reprodPossible2 == true){
		C_AQUARIUM* L_leAquarium;
		L_leAquarium = C_AQUARIUM::getAquarium();

		//On détermine le code génétique du fils
		unsigned int nbGene;
		nbGene = L_leAquarium->getTailleGenome();
		vector<int> codeGenetiqueFils;
		codeGenetiqueFils.resize(nbGene);
		vector<int> codeGenetiquePartenaire;
		codeGenetiquePartenaire = P_unSpeedbrick->getCodeGenetique();
		for(unsigned int cptGene=0; cptGene<nbGene; cptGene++){
			codeGenetiqueFils.begin();
			this->codeGenetique.begin();
			codeGenetiquePartenaire.begin();
			codeGenetiqueFils[cptGene] = this->codeGenetique[cptGene] +
				codeGenetiquePartenaire[cptGene];
		}

		//on cherche à savoir s'il va être NORMAL, GENTIL ou AGA
		//on cherche la différence entre le gene le plus grand et le second
		//plus grand
		int maxGene;
		int secondMaxGene;
		maxGene = 0;
		secondMaxGene = 0;
		for(unsigned int cptGene=0; cptGene < nbGene; cptGene++){
			codeGenetiqueFils.begin();
			if(maxGene < codeGenetiqueFils[cptGene]){
				maxGene = codeGenetiqueFils[cptGene];
			}
		}
		for(unsigned int cptGene=0; cptGene < nbGene; cptGene++){
			codeGenetiqueFils.begin();
			if((secondMaxGene < codeGenetiqueFils[cptGene]) && (maxGene >
				codeGenetiqueFils[cptGene])){
				secondMaxGene = codeGenetiqueFils[cptGene];
			}
		}

		unsigned int differenceGene;
		differenceGene = maxGene - secondMaxGene;

		if(differenceGene < 2){
			C_SPEEDBRICK_NORMAL* L_nouveauSpeedbrick;
```

```
                        L_nouveauSpeedbrick = new C_SPEEDBRICK_NORMAL(codeGenetiqueFils,
                                this->getPosX(), this->getPosY(), this->getPosZ());
                        C_SPEEDBRICK* L_unOrganisme;
                        L_unOrganisme = L_nouveauSpeedbrick->getInstanceSpeedbrick();
                        L_unOrganisme->ajouterOrganisme();
                        cout<<"Bébé speedbrick normal né avec le gènome : ";
                        L_unOrganisme->afficherCodeGenetiqueConsole();
                        cout<<endl;
                } else if(differenceGene < 3){
                        C_SPEEDBRICK_GENTIL* L_nouveauSpeedbrick;
                        L_nouveauSpeedbrick = new C_SPEEDBRICK_GENTIL(codeGenetiqueFils,
                                this->getPosX(), this->getPosY(), this->getPosZ());
                        C_SPEEDBRICK* L_unOrganisme;
                        L_unOrganisme = L_nouveauSpeedbrick->getInstanceSpeedbrick();
                        L_unOrganisme->ajouterOrganisme();
                        cout<<"Bébé speedbrick gentil né avec le gènome : ";
                        L_unOrganisme->afficherCodeGenetiqueConsole();
                        cout<<"."<<endl;
                } else if(differenceGene < 5){
                        C_SPEEDBRICK_AGA* L_nouveauSpeedbrick;
                        L_nouveauSpeedbrick = new C_SPEEDBRICK_AGA(codeGenetiqueFils,
                                this->getPosX(), this->getPosY(), this->getPosZ());
                        C_SPEEDBRICK* L_unOrganisme;
                        L_unOrganisme = L_nouveauSpeedbrick->getInstanceSpeedbrick();
                        L_unOrganisme->ajouterOrganisme();
                        cout<<"Bébé speedbrick aga né avec le gènome : ";
                        L_unOrganisme->afficherCodeGenetiqueConsole();
                        cout<<"."<<endl;
                }
        }
}
```

Scission

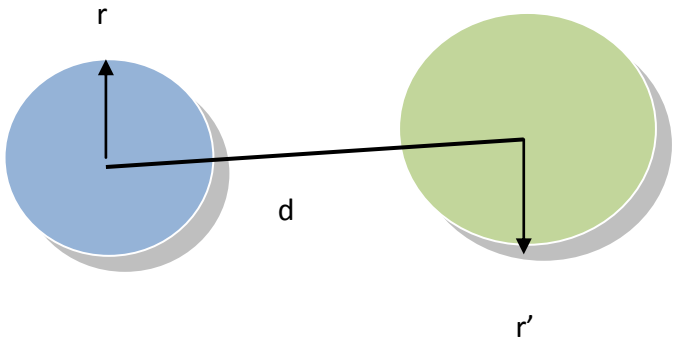
La reproduction par scission permet au Speedbrick de se multiplier en cas de contact avec un Cutit. Le nouveau Speedbrick étant une copie conforme de son « père » (ou de sa « mère »), nous avons décidé de mettre en place un système de constructeur par copie. Nous transférons ensuite les anneaux scindés vers le nouveau Speedbrick.

```
void C_SPEEDBRICK_NORMAL::seScinder(int P_positionAnneau){
        //si le speedbrick peut se faire couper
        if(this->dureeInvincible == 0){
                //on vérifie que le speedbrick possède assez d'anneaux
                int L_tailleListeAnneau;
                L_tailleListeAnneau = this->listeAnneaux.size();
                if(L_tailleListeAnneau >= P_positionAnneau){
                        C_SPEEDBRICK_NORMAL nouveauSpeedbrick;
                        nouveauSpeedbrick = *this;
                        C_SPEEDBRICK* ptrNouveauSpeedbrick;
                        ptrNouveauSpeedbrick = nouveauSpeedbrick.getInstanceSpeedbrick();
                        vector<C_ANNEAU*> listeAnneauxNouveauSpeedbrick;
                        listeAnneauxNouveauSpeedbrick = ptrNouveauSpeedbrick->getListeAnneaux();
                        listeAnneauxNouveauSpeedbrick.clear();
                        //tous les anneaux à partir de celui entré en collisions appartiennent au nouveau
speedbrick
                        for(int cptAnneau = P_positionAnneau; cptAnneau < L_tailleListeAnneau;
cptAnneau++){
                                listeAnneauxNouveauSpeedbrick.begin();
                                C_ANNEAU* pointeurSurAnneau;
                                pointeurSurAnneau = this->listeAnneaux[cptAnneau];
                                pointeurSurAnneau->setMonSpeedbrick(ptrNouveauSpeedbrick);
                                listeAnneauxNouveauSpeedbrick.push_back(pointeurSurAnneau);
                        }
                        ptrNouveauSpeedbrick->genererVecteurDeplacement();
                        ptrNouveauSpeedbrick->ajouterOrganisme();
                        for(int cptAnneau = L_tailleListeAnneau; cptAnneau > P_positionAnneau; cptAnneau--
){
                                this->listeAnneaux.pop_back();
                        }
                }
        }
}
```

Beau Morgan
Courazier Nicolas

Les collisions

Les Speedbrick étant formés de sphères, notre algorithme de collision détecte les collisions via des sphères virtuelles selon le schéma suivant :



Nous avons donc l'équation simplifiée suivante :

Si $d \leq d - r - r'$

Alors il y a collision

Exemple de test de collision entre un Speedbrick et un Cutit :

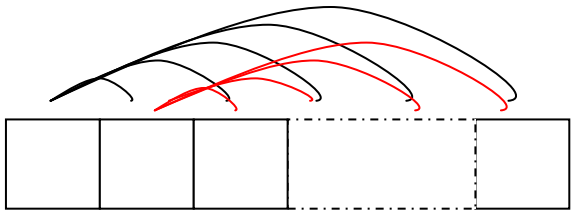
```
void testCollision(C_SPEEDBRICK* P_leSpeedbrick, C_CUTIT* P_leCutit){
	C_AQUARIUM* L_leAquarium;
	L_leAquarium = C_AQUARIUM::getAquarium();
	vector<C_ANNEAU*> L_listeAnneaux;
	L_listeAnneaux = P_leSpeedbrick->getListeAnneaux();

	if(L_listeAnneaux.size() != 0){
		C_ANNEAU* L_anneau;
		L_anneau = L_listeAnneaux[0];

		float lDistanceX = P_leCutit->getPosX() - P_leSpeedbrick->getPosX();
		float lDistanceY = P_leCutit->getPosY() - P_leSpeedbrick->getPosY();
		float lDistanceZ = P_leCutit->getPosZ() - P_leSpeedbrick->getPosZ();
		float lDistance = sqrt(lDistanceX*lDistanceX + lDistanceY*lDistanceY +
lDistanceZ*lDistanceZ);

		//si la tête est touchée, le speedbrick est coupé au milieu de ces anneaux
		if((lDistance - 8 - 3) <= 0){
			int L_anneau;
			L_anneau = (P_leSpeedbrick->getListeAnneaux().size()) / 2;
			L_leAquarium->interaction(P_leSpeedbrick, L_anneau);
		}

		L_listeAnneaux = P_leSpeedbrick->getListeAnneaux();
		for(unsigned int cptAnneaux = 0; cptAnneaux < L_listeAnneaux.size();
cptAnneaux++){
			if(P_leSpeedbrick->getDureeInvincible() == 0){
				L_anneau = L_listeAnneaux[cptAnneaux];
				float lDistanceX = P_leCutit->getPosX() - L_anneau->getPosX();
				float lDistanceY = P_leCutit->getPosY() - L_anneau->getPosY();
				float lDistanceZ = P_leCutit->getPosZ() - L_anneau->getPosZ();
				float lDistance = sqrt(lDistanceX*lDistanceX +
lDistanceY*lDistanceY + lDistanceZ*lDistanceZ);

				if((lDistance - 6 - 3) <= 0){
					L_leAquarium->interaction(P_leSpeedbrick, L_anneau-
>getIdAnneau());
				}
			}
		}
	}
}
```

Afin d'éviter toute redondance dans les tests de collision, c'est-à-dire que deux éléments soient comparés deux fois entre eux, nous avons mis en place deux boucles *pour* imbriquées.

Exemple pour les collisions Speedbrick / Speedbrick :

```
//GESTION DES COLLISIONS SP/SP
for (unsigned int cptSpeedbricks = 0; cptSpeedbricks < L_listeSpeedbricks.size()-1;
cptSpeedbricks++){
	C_SPEEDBRICK* L_leSpeedbrick;
	L_leSpeedbrick = L_listeSpeedbricks[cptSpeedbricks];
	for (unsigned int cptSpeedbrick2 = cptSpeedbricks+1 ; cptSpeedbrick2 <
	L_listeSpeedbricks.size(); cptSpeedbrick2++){
		C_SPEEDBRICK* L_leSpeedbrick2;
		L_leSpeedbrick2 = L_listeSpeedbricks[cptSpeedbrick2];
		testCollision(L_leSpeedbrick, L_leSpeedbrick2);
	}
}
```

Les comparaisons ont ainsi lieu de la manière suivante :



Liste des Speedbricks

Annexes

La charte d'écriture

Le nom des classes est noté en majuscule. Le préfixe « C_ » précède les concepts. Le préfixe « T_ » précède les classes d'énumération.

Les noms de méthodes et des attributs de classes commencent par une minuscule. Chaque composante de la variable commence par une majuscule.

Afin de déterminer simplement la portée de chacune des variables, elles sont préfixée par une lettre suivie du caractère « _ ».

Variable locale : L_variableLocale

Variable paramètre : P_variableParametre

Variables globales : G_variableGlobale

Le champ *this* est utilisé de manière explicite afin d'améliorer la lisibilité.

Qualité logicielle

Afin de respecter les règles de qualité logicielles, n'est effectuée dans la mesure du possible qu'une seule instruction par ligne.

Exemple :

~~Int champ = this->getValeurChamp() ;~~

```
Int champ ;
Champ = this->getValeurChamp() ;
```

Le C++ ne possédant pas de garbage collector, une attention toute particulière a été apportée à l'élimination de toute fuite mémoire, notamment dans les vector contenant des pointeurs vers des objets.