SeaTech École d'ingénieurs

Modélisation et Calculs Fluides et Structures

Compte Rendu de TP C++:

Programmation du jeu Casse-Brique

Enseignant: M.Impagliazzo

Etudiant: Dupont Ronan

Année Universitaire 2019-2020

Contents

Introduction

Dans ce TP, nous allons coder le jeu casse-brique en C++. L'avantage de ce langage est qu'il est orienté objet. Ceci permettra de faciliter les choses en codant le jeu étapes par étapes.
Sur ce rapport, nous allons expliquer la suite du jeu à partir du point où nous nous étions arrêtés en classe. Nous avions seulement une raquette ainsi que des balles avec les collisions : ball/murs et ball/raquette qu'avaient été gérés.

1 Collision ball/ball:

1.1 Détection

Pour la collision entre les balles, nous détecterons ceci d'une manière simple: lorsque la distance entre deux balles distinctes sera inférieure à $2R$ alors, on effectuera le traitement de la collision. Algorithmiquement ceci ce traduit de la manière suivante:

```
Point p1p2 = *(ballon->position) - *(this->position);
double distance = p1p2.norm();
if(distance <= this->radius*2.0){
        EFFECTUER TRAITEMENT
}
```

Pour ce qui est d'avoir deux balles indépendantes, nous lancerons deux boucles for dans la simulation avec la deuxième commençant à l'indice j=i+1 (i etant la première boucle allant de 0 au nombre de balles)

1.2 Traitement

Pour le traitement de la collision, nous nous inspirerons de la fonction collisionRacket qui prend en compte le fait que la balle tape sur l'angle de la raquette. Celui-ci était écrit de cette manière:

```
void World::collisionRacket(Ball* ballon){
 double r = this->ballRadius;
 double h = this->racketHeight;
 Point q = ballon->getPosition();
 Point p = this->racket-> getPosition();
 Point v = ballon->getVitesse();
 double l = this->racket->getCurrentWidth();
  if((q.y > h) && (q.x >p.x+0.5*l) &&(q.x < p.x+0.5*l +r)){
   Point c(p.x+0.5*l, p.y + 0.5*h);
   Point n = (q-c).normalize();
   if(v.dotProduct(n) < 0){
    Point t(-n.y, n.x);
        ballon->rebondAngulaire(n,t);
   }
  }else if((q.y > h) && (q.x < p.x-0.5*l) &&(q.x > p.x-0.5*l
     - r)){
   Point c(p.x - 0.5*l, p.y  + 0.5*h);
   Point n = (q-c).normalize();
   if(v.dotProduct(n) < 0){
    Point t(-n.y, n.x);
        ballon->rebondAngulaire(n,t);
```

```
    }
   }
}
```

Avec la fonction `rebondAngulaire` dans le `Ball.cpp`:

```
void Ball::rebondAngulaire(Point nn,Point tt){
    Point Vout = tt.scaledBy(this->vitesse->dotProduct(tt)) +
       nn.scaledBy(-this->vitesse->dotProduct(nn));
    this->vitesse->x = Vout.x;
    this->vitesse->y = Vout.y;
}
```

Les étapes pour obtenir un "rebond angulaire" il est nécessaire d'effecture un changement de repère et d'effectuer des projections (produit scalaire). Les étapes sont donc :

- Calculer un vecteur normal $\vec{n}$ unitaire entre les deux objets
- Calculer le vecteur tangente $\vec{t}$ unitaire qui est perpendiculaire à $\vec{n}$.
- Passer l'objet dans la fonction `rebondAngulaire` qui calculera les nouvelles vitesses V_x et V_y.

Pour résumer ceci, nous avons codé la fonction `collisionBall` dans `ball.cpp` (ceci permet de pouvoir changer les paramètres de deux balles tout en n'en prennant qu'une en argument):

```
void Ball::collisionBall(Ball* ballon){
 Point p1p2 = *(ballon->position) - *(this->position);
 double distance = p1p2.norm();
 if(distance <= this->radius*2.0){
  Point n = p1p2.normalize();
  Point t (-n.y,n.x);
  if(ballon->vitesse->dotProduct(*this->vitesse) <0.0){
   ballon->rebondAngulaire(n,t);
   this->rebondAngulaire(n,t);
  }
 }
}
```

On lance donc cette fonction dans la simulation de la manière suivante:

```
//collision ball-ball
for(int i=0;i<this->usedBall;i++){
 if(this->ball[i]->getState() == Ball::FREE){
  for(int j=i+1;j<this->usedBall;j++){
   if(this->ball[j]->getState() == Ball::FREE){
    this->ball[i]->rebondBall(this->ball[j]);
   }
  }
 }
}
```

2 Génération de briques :

Pour générer des briques, nous nous sommes très fortement inspiré des classes racket et ball.

Nous avons dans un premier temps effectué une copie des fichiers racket.cpp et racket.hpp. On a ensuite tout simplement changé tous les "racket" par "brique". **On pensera à retirer la fonction move car nos briques ne doivent pas bouger.**// Ceci nous faisait déjà un progrfonction mamme fonctionnel. On a ensuite ajouté des états comme pour la classe ball:

```
const int Brique::DURETE2 = 2;
const int Brique::DURETE1 = 1;
const int Brique::OUT =0;
```

Il était ensuite nécessaire que les briques ne se génèrent pas au même endroit que la raquette. C'est pourquoi nous avons édité la fonction de création de brique:

```
Brique::Brique(double window_width,double
   initial_Brique_width,double Brique_height,double
   Brique_zone_height){
this->position = NULL;
double rds=rand()%100;
rds=rds/50+1;
rds=int(rds); //Aleatoire soit 1 soit 2
this->state = rds;
this->initialWidth = initial_Brique_width;
this->currentWidth=initial_Brique_width;
this->thickness = Brique_height;
double rdx=rand()%900+50;
rdx=rdx/1000; //nb entre 0.05 et 0.95
double rdy=rand()%900+50;
rdy=rdy/1000; //nb entre 0.05 et 0.9
this->position = new Point(rdx*window_width,
   Brique_zone_height+250*rdy);
this->windowWidth = window_width;
}
```

On peut voir que nous avons rajouté un argument "Brique_zone_height" (il a donc fallu le rajouter dans l'argument de la construction du monde) qui permet de de savoir la taille de la zone où les zones seront generés.
En observant le code ci-dessous, on remarque que l'état de dureté est generé aléatoirement. De même pour l'emplacement de la brique qui est généré aléatoirement dans la zone.

Ensuite à la même manière que la création de ball, nous créons des briques dans des tableaux. On déclare donc dans `world.hpp` le tableau en donnée privée : `Brique** brique;`. Et donc dans la fonction création de brique sera:

```
this->brique = new Brique*[2*this->levelNumber];
for(int i=0; i<2*this->levelNumber; i++){
 this->brique[i] = new Brique(window_width,
    initial_brique_width,brique_height,brique_zone_height);
}
```

Le destructeur de brique sera lui (dans `world:W̃orld()`:

```
if(this->brique != NULL){
 for(int i=0; i<2*this->levelNumber; i++){
  delete this->brique[i];
 }
 delete [] this->brique;
```

```
}
```

Puis ensuite la fonction de création de monde deviendra:

```
void World::draw(){
 this->racket->draw();          // Dessin de la racket
 for(int i=0; i<this->currentLevel; i++){
  this->ball[i]->draw();
 }
 for(int i=0; i<2*this->currentLevel; i++){
  this->brique[i]->draw();
 }
}
```

3 Collision ball/brique:

Il est donc nécessaire d'ensuite gérer la collision entre les balles et les briques.

3.1 Détection

Pour se faire, nous allons fortement nous inspirer de la collision racket/ball.
Etant donné que les vitesses Vx, Vy ne changent pas de la même manière en fonction du côté de la brique où la balle tape, nous avons effectué une detection par côté de la manière suivante:

```
// rebonds bas: x fixe et y change
if((p.x>=(r.x-initialBriqueWidth/2-R)) && (p.x<=(r.x+
   initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2-R))
   && (p.y<=(r.y-briqueHeight/2+R)))

// rebonds haut
if((p.x>=(r.x-initialBriqueWidth/2-R)) && (p.x<=(r.x+
   initialBriqueWidth/2+R)) && (p.y>=(r.y+briqueHeight/2-R))
   && (p.y<=(r.y+briqueHeight/2+R)))

// rebonds gauche: y fixe et x change
if((p.x>=(r.x-initialBriqueWidth/2-R)) && (p.x<=(r.x-
   initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2)) &&
    (p.y<=(r.y+briqueHeight/2)))

// rebonds droite
if((p.x>=(r.x+initialBriqueWidth/2-R)) && (p.x<=(r.x+
   initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2)) &&
    (p.y<=(r.y+briqueHeight/2)))
```

3.2 Traitement

Il était ensuite nécessaire d'effectuer deux choses une fois la collision détecté:

- Renvoyer la balle dans la bonne direction
- Diminuer la dureté de la brique (possibilité de la faire disparaitre)
- Ajouter des points au score

3.2.1 Rebond ball

Pour le premier point, on remarque que pour une collision sur une surface horizontale, les transformations de vitesses suivantes sont appliqués:
$V_x' = V_x$ et $V_y' = -V_y$
Ceci est donc représenté par la fonction rebond suivante:

```
void Ball::rebondBrique(){
        this->vitesse->y = -this->vitesse->y;
        this->vitesse->x = this->vitesse->x;
}
```

Pour la collision sur une surface verticale, la transformation effectuée est la suivante:
$V_x' = -V_x$ et $V_y' = V_y$
Ceci est donc représenté par la fonction rebond suivante:

```
void Ball::rebondBrique2(){
        this->vitesse->y = this->vitesse->y;
        this->vitesse->x = -this->vitesse->x;
}
```

3.2.2 Diminution de la dureté

Etant donné que nous avons correctement programmé les états de brique, nous avons pu créer une fonction changeant l'état de la brique:

```
void Brique::isOut(){
        this->state = this->getState()-1;
}
```

De plus, on pourra réutiliser la fonction `reduceWidth` de la raquette pour que la brique devienne d'une longueur nulle:

```
void Brique::reduceWidth(){
        this->currentWidth=this->currentWidth-1*this->
           currentWidth;
}
```

3.2.3 Ajout de points au score

Nous ajouterons tout simplement des points avec l'instruction:
`this->currentScore=this->currentScore +10;` à chaque collision entre la balle et une brique.

3.2.4 Bilan

Il ne nous reste plus qu'à assembler tout ça ensemble de la manière suivante:

```
void World::collisionBrique(Ball* ballon, Brique** brique){
 for(int i=0; i<this->levelNumber; i++){
  if(this->brique[i]->getState() == Brique::DURETE1 or this->
     brique[i]->getState() == Brique::DURETE2 ){
  Point p = ballon->getPosition();  //position ballon
  Point vitesse = ballon->getVitesse();
  Point r = brique[i]->getPosition();   //position
  int R= this->ballRadius;
```

```
  initialBriqueWidth=brique[i]->currentWidth;

  // rebonds bas: x fixe et y change
  if((p.x>=(r.x-initialBriqueWidth/2-R)) && (p.x<=(r.x+
     initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2-R)
     ) && (p.y<=(r.y-briqueHeight/2+R))){
  ballon->rebondBrique();

  if(this->brique[i]->getState() == Brique::DURETE1){
  this->brique[i]->reduceWidth();}
  this->brique[i]->isOut();
  this->currentScore=this->currentScore +10;}
  // rebonds haut
  if((p.x>=(r.x-initialBriqueWidth/2-R)) &&  (p.x<=(r.x+
     initialBriqueWidth/2+R)) && (p.y>=(r.y+briqueHeight/2-R)
     ) && (p.y<=(r.y+briqueHeight/2+R))){
  ballon->rebondBrique();

  if(this->brique[i]->getState() == Brique::DURETE1){
  this->brique[i]->reduceWidth();}
  this->brique[i]->isOut();
  this->currentScore=this->currentScore +10;}

  // rebonds gauche: y fixe et x change
  if((p.x>=(r.x-initialBriqueWidth/2-R)) && (p.x<=(r.x-
     initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2))
     && (p.y<=(r.y+briqueHeight/2))){
  ballon->rebondBrique2();

  if(this->brique[i]->getState() == Brique::DURETE1){
  this->brique[i]->reduceWidth();}
  this->brique[i]->isOut();
  this->currentScore=this->currentScore +10;}
  // rebonds droite
  if((p.x>=(r.x+initialBriqueWidth/2-R)) && (p.x<=(r.x+
     initialBriqueWidth/2+R)) && (p.y>=(r.y-briqueHeight/2))
     && (p.y<=(r.y+briqueHeight/2))){
  ballon->rebondBrique2();

  if(this->brique[i]->getState() == Brique::DURETE1){
  this->brique[i]->reduceWidth();}
  this->brique[i]->isOut();
  this->currentScore=this->currentScore +10;}
  }
 }
}
```

On ajoutera donc cette fonction dans la simulation du `world` à l'aide de l'instruction suivante:

```
//collision brique
for(int i=0; i<this->usedBall; i++){
 if(this->ball[i]->getState() == Ball::FREE){
```

```
  this->collisionBrique(this->ball[i],this->brique);
 }
}
```

4 Passage de niveaux

Pour passer les niveaux, nous créons une fonction `void checkBrique(Brique**);` dans world. Celle-ci va compter le nombre de brique cassés et comparer ce nombre au nombre total de briques génerés. Si ce nombre est égal, l'instruction de passer au niveau supplémentaire sera lancé. Il serait sûrement nécessaire de relancer une fonction draw dedans mais le fonction ne fonctionne pas pour le moment: `Erreur de segmentation (core dumped)`.

```
void World::checkBrique(Brique** brique){
 int nb=0;
 for(int i=0; i<2*this->levelNumber; i++){
  if(this->brique[i]->getState() == Brique::OUT){
   nb=nb+1;}
  }
  if(nb == 2*levelNumber){
   this->currentLevel = this->currentLevel +1;
  }
}
```

On lancera cette fonction dans la simulation:

```
this->checkBrique(this->brique);
```

Mais pour le moment avec la ligne de passage de niveau en commenté.

Conclusion