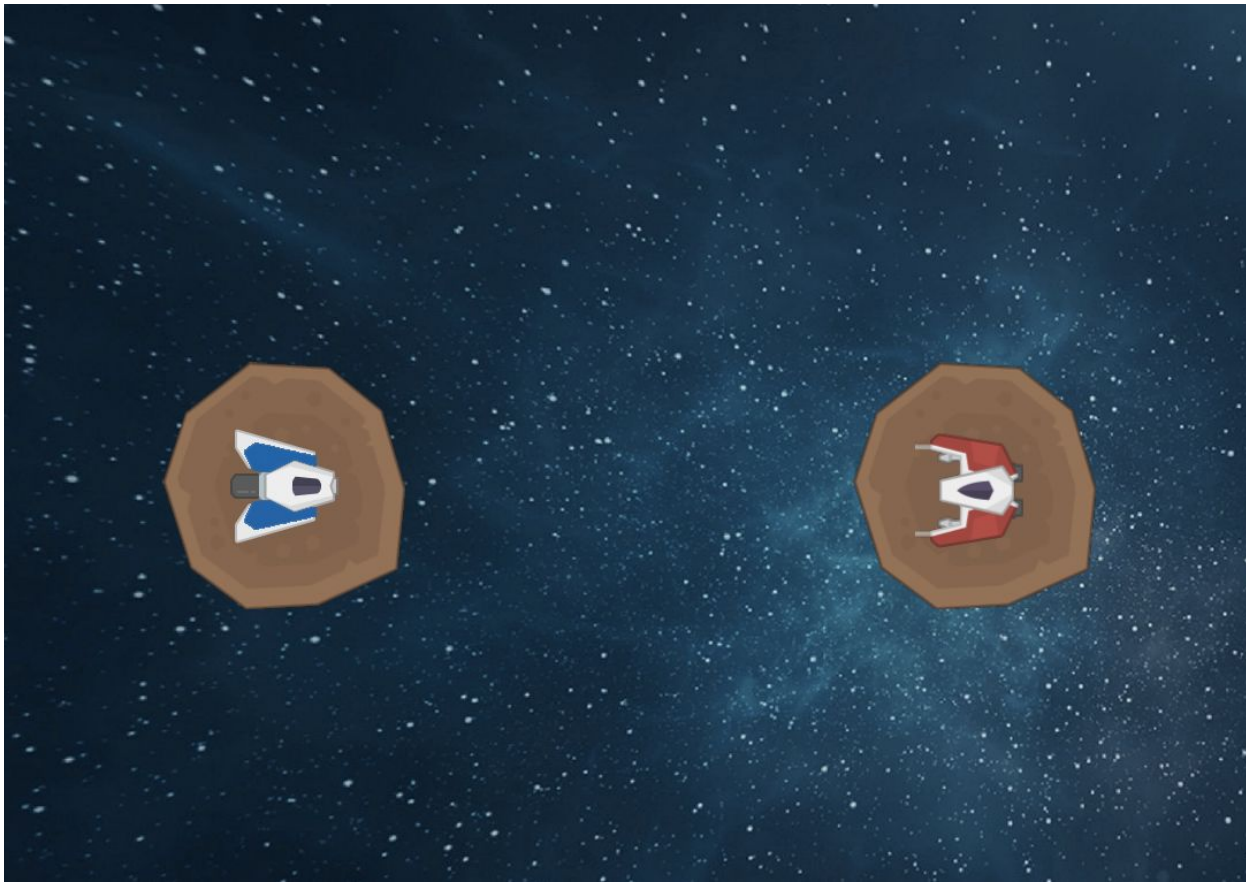
RAPPORT PROJET LIBRE

Création d'un mini jeu régi par les lois de la physique



Homero Restrepo

Ferdinand Tixidre

Étudiants en L3 de Sorbonne Sciences
UE LU3PY002

TABLE DES MATIERES

INTRODUCTION

Dans le cadre de l'UE LU3PY002 nous avons entrepris le projet [Nom du jeu]. L'objectif principal étant la création d'un mini jeu stratégique de guerre intergalactique à deux joueurs. Ce mini jeu a été inspiré du jeu M.A.R.S. Shooter (1), ce qui nous a intéressé dans ce jeu est le mouvement des missiles que tirent les vaisseau, régis par des champs gravitationnels des planètes présentes dans le cadre. L'utilisation de la bibliothèque SFML (Simple and Fast Multimedia Library) (2), est au coeur de ce projet, elle nous permet d'afficher une fenêtre, des objets, de modifier leur apparence ou encore de gérer les entrées clavier/souris.

Le jeu se joue à deux, l'objectif est de détruire le vaisseau de votre adversaire en tirant des missiles mais ces derniers seront attirés par de larges astres qui génèrent de forts champs de gravités. On peut donc se cacher derrière ces astres en espérant qu'ils absorbent les tirs ennemis ou utiliser l'effet de l'interaction gravitationnelle pour ajuster la trajectoire des missiles.

Le joueur 1 utilise les flèches du clavier pour se diriger et la touche espace pour tirer, le joueur 2 se dirige avec ZQSD et tire avec la touche F.

RESSOURCES

Nous avons utilisé des images libres de droits, trouvées sur le site opengameart.org (3) qui est mentionné dans la partie sur SFML du polycopié de cours.

La fonction rk4, pour Runge-Kutta d'ordre 4, est donnée dans les ressources du cours sur Moodle car elle est utilisée dans le TP4. Il s'agit d'une méthode d'approximation de solutions d'équations différentielles. Elle repose sur le principe d'itération, c'est à dire que la première estimation est utilisée pour calculer la deuxième, plus précise, et ainsi de suite.

```c
void rk4(int n, double x,double y[] ,double dx,
                void deriv(int, double, double[], double[]))
/*----------------------------------------
 sous programme de resolution d'equations
 differentielles du premier ordre par
 la methode de Runge-Kutta d'ordre 4
 x = abscisse
 y = valeurs des fonctions
 dx = pas
 n = nombre d'equations differentielles
 deriv = variable contenant le nom du
 sous-programme qui calcule les derivees
 ----------------------------------------*/
{
int i ;
double ddx ;
/* d1, d2, d3, d4 = estimations des derivees
   yp = estimations intermediaires des fonctions */
double d1[n], d2[n], d3[n], d4[n], yp[n];

ddx = dx/2;                 /* demi-pas */

deriv(n,x,y,d1) ;           /* 1ere estimation */

for( i = 0; i< n; i++){ yp[i] = y[i] + d1[i]*ddx ; }
deriv(n,x+ddx,yp,d2) ;      /* 2eme estimat. (1/2 pas) */

for( i = 0; i < n; i++){ yp[i] = y[i] + d2[i]*ddx ; }
deriv(n,x+ddx,yp,d3) ; /* 3eme estimat. (1/2 pas) */

for( i = 0; i< n; i++){ yp[i] = y[i] + d3[i]*dx ;}
deriv(n,x+dx,yp,d4) ;       /* 4eme estimat. (1 pas) */
/* estimation de y pour le pas suivant en utilisant
  une moyenne pondÈrÈe des dÈrivÈes en remarquant
  que : 1/6 + 1/3 + 1/3 + 1/6 = 1 */

for( i = 0; i < n ; i++)
 { y[i] = y[i] + dx*( d1[i] + 2*d2[i] + 2*d3[i] + d4[i] )/6 ; }
}
```

LOIS PHYSIQUES

1. Mouvement des vaisseaux

Notre premier objectif était de pouvoir déplacer les vaisseau affichés, les explications concernant l'affichage liés à SFML sont dans la troisième partie, (Structure du programme). Les déplacements des vaisseaux sont régies par des équations différentielles qui prennent en compte les dérivées secondes et premières de la position par rapport au temps.

$$\mathbf{r} = \mathbf{r}(t)$$

$$\mathbf{v} = \frac{d\boldsymbol{r}}{dt}$$

$$\mathbf{a} = \frac{d\boldsymbol{v}}{dt} = \frac{d^2\boldsymbol{r}}{dt^2}$$

On crée donc pour chaque dimension un tableau de 3 éléments le premier est la position, le second la vitesse et le troisième l'accélération.

```
105         void SetPosition(double xPos,double yPos){
106             this->x[0] = xPos;
107             this->y[0] = yPos;
108         }
109
110         void SetSpeed(double xSpeed,double ySpeed){
111             this->x[1] = xSpeed;
112             this->y[1] = ySpeed;
113         }
114         void SetForces(double xForce,double yForce){
115             this->x[2] = xForce;
116             this->y[2] = yForce;
117         }
```

« Tout corps persévère dans l'état de repos ou de mouvement uniforme en ligne droite dans lequel il se trouve, à moins que quelque force n'agisse sur lui, et ne le contraigne à changer d'état. »

Selon l'énoncé de la première loi de Newton, notre vaisseau spatial doit garder sa vitesse constante et son mouvement en ligne droite après une première accélération. Cependant nous allons ajouter l'équivalent d'une force de frottement qui va ralentir le vaisseau afin de rendre son contrôle plus simple.

```
345 class ship: public space_object{
346
347 private:
348
349     void dynamics(int n, double t, double y[], double dy[])
350     {
351         dy[0] = y[1];
352         dy[1] = y[2] - 0.45*y[1]; // -y[1] est une force non conservative pour faciliter le controle du
                vaisseau
353     }
```

Pour l'instant le vaisseau ne déplace qu'en ligne droite, on peut le faire pivoter sur lui même en utilisant les fonctions trigonométriques. C fonctionne avec des radians et on fait la conversion degré en radians en multipliant par 0.017453.

```
130       double GetDirectionX() { return cos( phy*0.017453f );}
131       double GetDirectionY() { return sin( phy*0.017453f );}
```

On définit une vitesse maximale au-delà de laquelle les vaisseaux ne peuvent pas accélérer, afin de garder les vaisseau contrôlable et le jeu lisible.

```
108         //Deffines Speed limits
109         if ( x[1] > abs(MaxSpeed) ){
110             x[1]=MaxSpeed;
111         }
112
113         if ( y[1] > abs(MaxSpeed) ){
114             y[1]=MaxSpeed;
115         }
116
117         if ( x[1] < -abs(MaxSpeed) ){
118             x[1]=-MaxSpeed;
119         }
120
121         if ( y[1] < -abs(MaxSpeed) ){
122             y[1]=-MaxSpeed;
123         }
124
```

Les limites de la fenêtre étaient problématiques car on pouvait perdre la vaisseau de vue, il a donc fallu trouver une façon de garder le vaisseau dans la fenêtre. Notre solution est d'assimiler l'espace de jeu à un tore, ainsi quand le vaisseau passe par un bord de la fenêtre il réapparaît de l'autre côté.

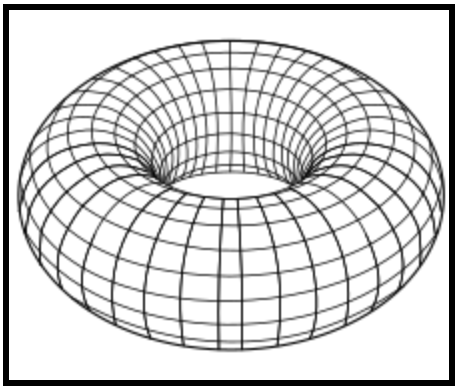


Illustration d'un tore (source : Wikipedia)

Ce qui se traduit en C par les conditions suivantes sur les coordonnées x et y.

```
89              //Deffines Spacial limits
90              if (x[0] > windowSizeX) x[0]=0; if (x[0] < 0) x[0]=windowSizeX;
91              if (y[0] > windowSizeY) y[0]=0; if (y[0] < 0) y[0]=windowSizeY;
```

2. Force gravitationnelle

La loi de gravitation universelle de Newton permet décrit la gravitation comme une force responsable du mouvements des objets célestes, de l'attraction entre les corps ayant une masse. La force F qu'exerce le corps A sur le corps B séparés par une distance d s'écrit ainsi :

$$F_{A/B} = G\frac{M_A M_B}{d^2}$$

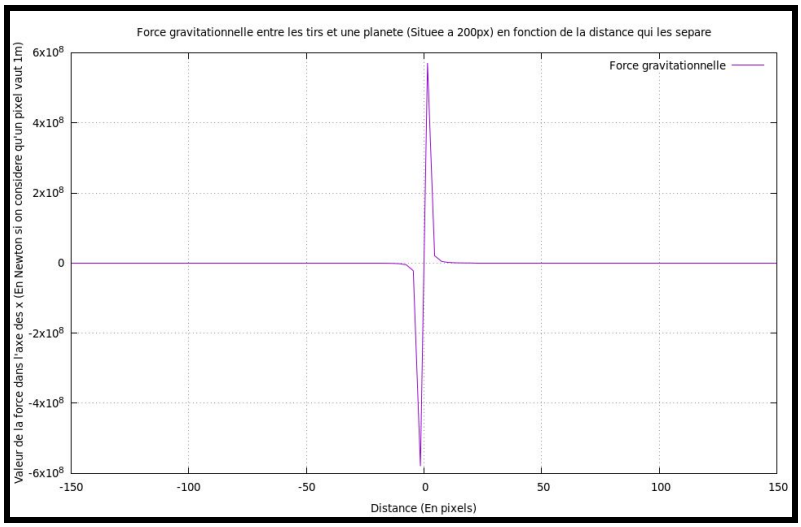
Où G est la constante gravitationnelle telle que :

$$G = 6,67408 \times 10^{-11} N \cdot m^2 \cdot kg^{-2}$$

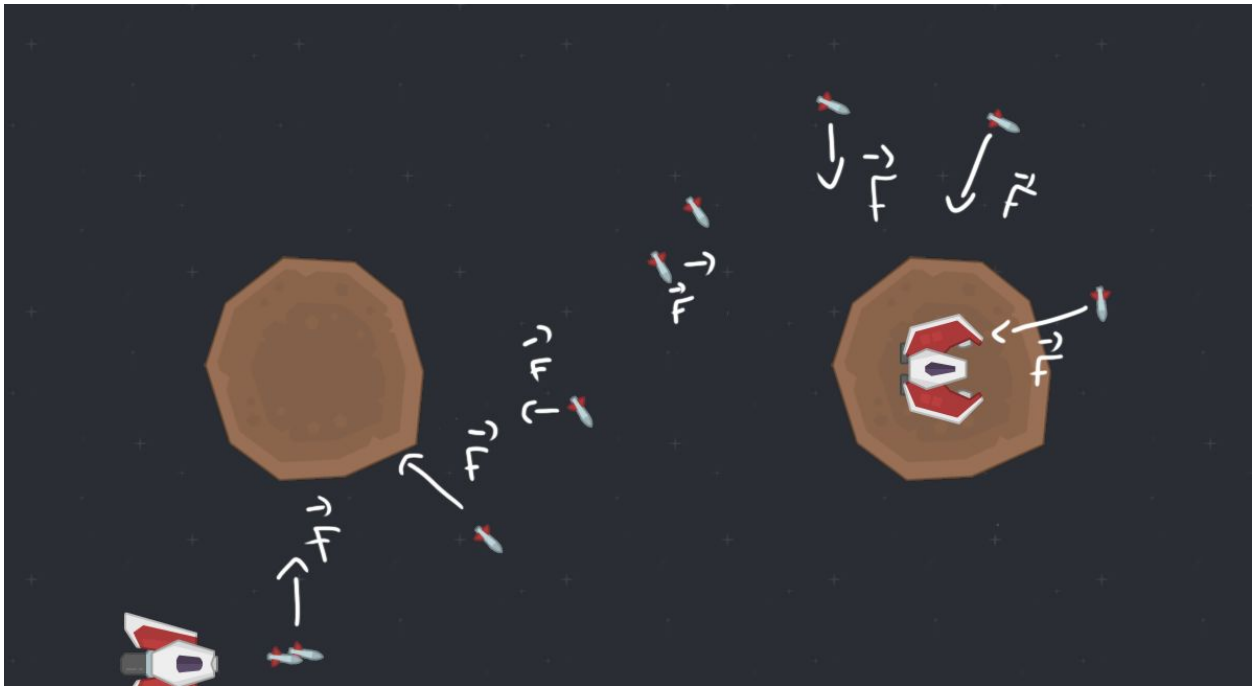
```
284
285     void externalForce(space_object& a){
286
287         x[2] += a.gravity * (a.x[0]-x[0])*pow(distance(a),-3);
288         y[2] += a.gravity * (a.y[0]-y[0])*pow(distance(a),-3);
289
290     }
291
```

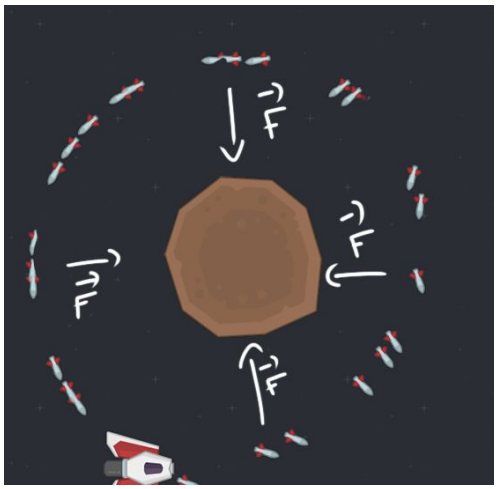
Dans le int main(){...} On a fixé le coefficient gravity des forces à la valeur de 1e7 pour toutes les planètes et missiles. Ce coefficient est égale au produit des masses fois la constante gravitationnelle. Ce qui comparé avec le champ gravitationnel lunaire nous permettrait de déduire la masse équivalente qui suivrait la même force que notre missile si elle se déplacerait à moins d'un kilomètre du centre de l'astre. Cette analogie nous permet de dire que si l'astéroïde aurait la masse de la lune alors le missile aurait une masse d'environ 2mg. Ce coefficient nous permet d'avoir un effet de gravité assez esthétique dans des distances de quelques centaines de pixels. Ceci est bien sur un choix personnel.

L'évolution de la composante de la force gravitationnelle dans les axes des Y en fonction de la distance est donné dans le graphique suivant. La composante selon l'axe des X a un comportement analogue donc on n'a pas tracé son évolution. Les axes des Y sont tracés de façon opposé à ce qui est fait conventionnellement (L'axe est dirigé vers le bas).



Donc, on voit bien que le missile est toujours attiré vers la planète.





On a reproduit ici schématiquement les forces gravitationnelles exercées sur les tirs.

STRUCTURE DU PROGRAMME

Pour la structure du jeu on a décidé d'utiliser la programmation orientée objet pour ne pas avoir à définir des fonctions identiques pour chaque objet créé.

1. Classe Space_object

Dans Cette Classe on a défini toutes les variables et fonctions (Méthodes) Communes à tous les objets présents dans le jeu. Notamment on a définit la fonction qui s'occupe de calculer les positions des objets en résolvant le système différentiel qui définit leur mouvement. Cette fonction est rk4 et elle calcule les positions suivantes en fonction des positions, des vitesses, et des forces avec une précision à 1e-4 près. Elle prend comme arguments le rang du système différentiel, le point en abscisse considéré (Ici on va le fixer arbitrairement car on va définir la dépendance spatiale des forces dans une autre fonction), les conditions initiales présentes dans un vecteur de la même taille que le rang du système différentiel, et le pas qui définit avec quelle précision on fera l'approximation des solutions (ici les positions et vitesses suivantes).

On utilise cette fonction en conjonction avec applyLimits() pour définir le mouvement des objets présents dans l'espace. La fonction applyLimits se charge de vérifier les conditions limites du jeu. En particulier elle permet de positionner le vaisseau de l'autre côté de la fenêtre quand il dépasse le cadre.

Lorsque les nouvelles coordonnées vérifient bien les conditions de position et de vitesse elles sont attribués aux sprites qui vont représenter graphiquement l'objet.

D'autres fonctions qui servent à afficher les données des variables ou à modifier les valeurs de celles-ci sont présentes dans cette classe. Elles nous ont permis d'observer l'évolution des positions, vitesses et forces de différents objets.

2. Classe Ship

On crée deux classes ship et ship2, une pour chaque joueur. Elles héritent des méthodes et variables de la classe Space_object. C'est dans cette classe que sont définis les conditions pour tirer (nombres de missiles maximum à l'écran, temps entre chaque tir). On a utilisé l'horloge de la bibliothèque chrono pour acquérir l'évolution temporelle en ms depuis le début de l'exécution. On utilise cette horloge pour calculer les évolutions temporelles et mettre un rythme entre chaque tir.

On a stocké tous les tirs lancés dans le Vector<Shot> shotsInSpace et on les a effacés dès qu'ils ne vérifient plus les conditions de vie. Par exemple lorsqu'ils ont dépassé un certain temps de vie ou qu'ils ont rentré en collision avec un autre objet.

3. Classe Ship 2

Cette classe hérite toutes les variables et les méthodes de la classe ship. On a juste changé la fonction GetInput qu'on avait définie comme virtuelle dans ship pour que le deuxième joueur 2 puisse avoir des touches différentes.

4. Classe planet

Dans cette classe on a juste créé la méthode RandPosition Dans l'hypothèse ou on aurait voulu que les planètes aient un mouvement aléatoire dans le temps. On a observé que cette fonctionnalité supplémentaire ajoutait des difficultés qui rendaient le jeu injouable. Ceci est un choix qu'on a effectué mais le mouvement des planètes reste possible.

5. SFML

On souhaite afficher une fenêtre et laisser aux joueurs la liberté de choisir la taille de celle-ci et leur nom dans le jeu ce qui se fait simplement avec les fonctions cout et cin.

```
int main(){

    /// Menu ....

        int windowSizeX = 1200, windowSizeY = 700;

        string name1,name2;

        cout << "\nInserez le nom du joueur N°1: ";cin >> name1;
        cout << "\nInserez le nom du joueur N°2: ";cin >> name2;

        cout << "\nInserez la taille horizontale de la fenetre de jeu: ";   cin >> windowSizeX;
        cout << "\nInserez la taille verticale de la fenetre de jeu: ";     cin >> windowSizeY;

    /// PROPRIETES DU FOND ....


        RenderWindow window(VideoMode(windowSizeX, windowSizeY), "Spacecraft Movement");
        window.setFramerateLimit(40);
        Texture t1;
        t1.loadFromFile("blue.png");
        t1.setRepeated(true);
        Sprite sFond(t1,IntRect(0,0,windowSizeX,windowSizeY));
```

L'affichage de la fenêtre se fait avec une boucle while, ainsi la fenêtre reste ouverte tant qu'on ne lui demande pas de se fermer.

```
while (window.isOpen()){

    Event event;
    while (window.pollEvent(event)){
        if (event.type == Event::Closed) window.close();
      }
    //Closses the window when you press (x)

    //Sets the background
            window.clear(Color::Black);
    window.draw(sFond);

    //clears the terminal
    system("clear");
```

SFML permet de gérer les entrées du clavier. On va utiliser les flèches du clavier et la touche espace pour le premier joueur et les touches Z,Q,S,D plus la touche F pour le second. On les assigne à différents mouvement du vaisseau ainsi :

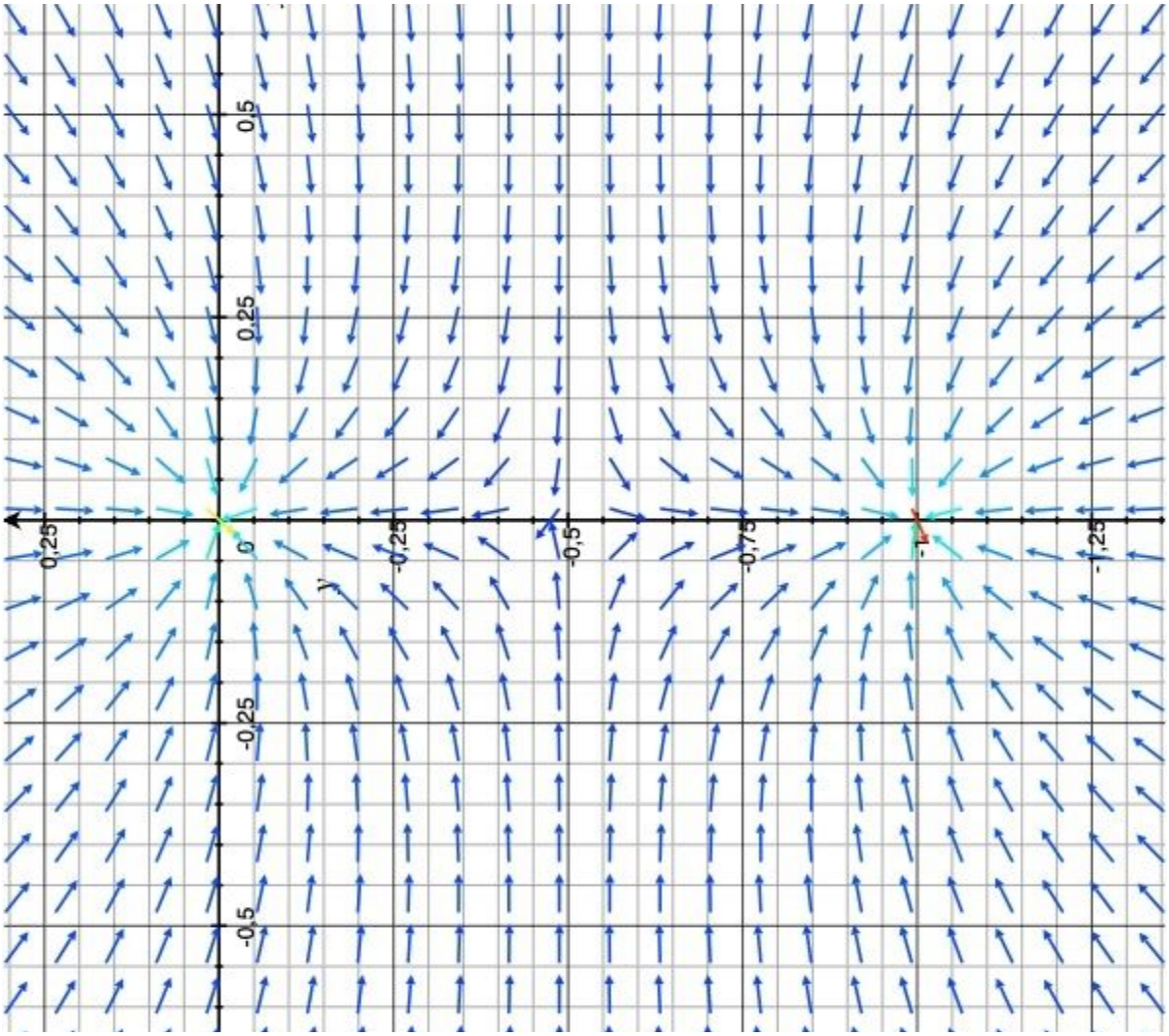
```
420     void ship::GetInput(int sensibility){
421
422             if ( Keyboard::isKeyPressed(sf::Keyboard::Left) )  {        this->phy   += -sensibility;
423                 if ( Keyboard::isKeyPressed(sf::Keyboard::Up) ){        this->trust += sensibility;
424                     if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
425                     else firing = false;
426                 }
427                 else if ( Keyboard::isKeyPressed(sf::Keyboard::Down) ){ this->trust += -sensibility;
428                     if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
429                     else firing = false;
430                 }
431                 else if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
432                 else firing = false;
433             }
434
435             else if ( Keyboard::isKeyPressed(sf::Keyboard::Right) ) {   this->phy   += +sensibility;
436                 if ( Keyboard::isKeyPressed(sf::Keyboard::Up) ){         this->trust += sensibility;
437                     if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
438                     else firing = false;
439                 }
440                 else if ( Keyboard::isKeyPressed(sf::Keyboard::Down) ){  this->trust += -sensibility;
441                     if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
442                     else firing = false;
443                 }
444                 else if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
445                 else firing = false;
446             }
447             else if ( Keyboard::isKeyPressed(sf::Keyboard::Up) ){        this->trust += sensibility;
448                 if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
449                 else firing = false;
450             }
451             else if ( Keyboard::isKeyPressed(sf::Keyboard::Down) ){      this->trust += -sensibility;
452                 if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
453                 else firing = false;
454             }
455
456             else if ( Keyboard::isKeyPressed(sf::Keyboard::Space) ) firing = true;
457             else {
458
459                 trust = 0.;
460                 this->GetVectorX()[2] = 0.;
461                 this->GetVectorY()[2] = 0.;
462                 ResetShotCooldown();
463                 firing = false;
464
465             }
```

On a aussi défini un angle en fonction de la vitesse de chaque tir à la ligne 774. Ceci permet de faire pivoter les missils sur eux mêmes vers la direction dans laquelle ils se déplacent:

$$\varphi = 180 + \frac{180}{\pi} acos\left(\frac{V_x}{\sqrt{{V_x}^2+{V_y}^2}}\right)$$

DONNÉES DU CHAMP GRAVITATIONNEL

Le champ gravitationnel généré par deux astres de même masse séparés d'une distance d (d=1 ci dessous) est tracé ci dessous.



On observe que ceci est bien en accord avec ce qu'on a trouvé dans la partie Lois physiques-2.

L'idée d'ajouter deux astéroïdes est qu'ils vont servir de refuge pour les vaisseaux parce qu'ils attirent les missiles. Cependant les astéroïdes ont une vie et ils peuvent être

détruits par l'ennemi. Ces objets permettent d'ajouter une difficulté stratégique au jeu.

RESULTATS

Les équations physiques entrant en compte dans notre programme sont correctement modélisées. Les mouvements des vaisseaux et des missiles sont cohérents. On observe que les trajectoires des missiles suivent bien les directions du champ gravitationnel théorique ci dessus. On est donc plutôt satisfaits des résultats.

CONCLUSION

La réalisation de ce jeu a demandé beaucoup de temps, elle fut très enrichissante d'un point de vue pédagogique car nous avons utilisé beaucoup d'éléments de C++ (utilisations des classes), dont nous avons dû comprendre le fonctionnement. On recommande aussi aux lecteurs de jouer à ce jeu il peut s'avérer très amusant.

Les ressources du jeu ainsi que l'exécutable peuvent se trouver dans le GitHub: https://github.com/Romeo75/C-Game-Project

REFERENCES

(1) - http://mars-game.sourceforge.net
(2) - https://www.sfml-dev.org
(3) - https://opengameart.org/content/space-shooter-redux
(4) - https://stackoverflow.com/questions/

ANNEXE

Code c++ employé page suivante.