Rapport_Projet_LE_MONTAGNER_ROBEYNS

Roman.Le Montagner, Matthieu.Robeyns

April 2020

Contents

1 Introduction

Bienvenue dans le rapport du Jeu de Le Montagner Roman et Robeyns Matthieu. Le jeu dont nous allons vous parler possède un comportement similaire au fameux jeux AngryBirds, c'est à dire, détruire des cibles via un nombre limité de projectiles que nous possédons. Nous allons, dans un premier temps, vous parler du coeur du sujet avec le moteur physique du jeu, aka la gestion des collision, gestion de force, (etc), qui a été entièrement coder par Le Montagner Roman. Puis nous parlerons de l'aspect logique du jeu, avec la gestion du menu, la lecture des niveaux et projectiles, qui a été codé par Robeyns Matthieu.

2 Moteur Physique

2.1 Point d'entré du moteur physique

Le point d'entré du moteur physique dans le programme se trouve dans le fichier "2DPhysicsEngine.js". Un objet de type "PhysicsEngine" est instancié par l'objet "GameLogic" permettant de géré la physique du jeu. Cette objet GameLogic contient la fonction qui est appelé à chaque frame par RequestAnimationFrame. Les deux fonctions qui permettent de géré le moteur physique à chaque frame sont donc appelé dans cette fonction de GameLogic et se nomme "renderEngine" et "applyPhysics".

La fonction "renderEngine" permet tous simplement de dessiné sur le canvas les objets géré par la physique. La seconde fonction "applyPhysics" applique les lois de la physique de notre jeu à chacun des objets. La fonction "renderEngine" étant assez simple, je n'entrerai pas dans les détails de son implémentation ici, nous nous attarderons plus particulièrement sur la fonction "applyPhysics".

La fonction applyPhysics suit l'automate très simple suivant :

- 1: Appliquer les lois de mouvements
- 2: Détection des collisions
- 3: Résolution des collisions
- 4: Retour en 1 au début de la prochaine frame

Nous expliciterons chacune des trois phases dans les sous-sections suivante :

2.2 Loi du mouvement

Nous allons dans cette section traité des lois de mouvements que nous souhaitons donné à nos objets. Nous nous appuyons pour cela sur les lois du mouvement de Newton. Chacun de nos objet soumis à la physique doivent hérité des objets de type "Dynamics" dont l'implémentation est donné dans le fichier "dynamics.js". Chacun des objets de type "Dynamics" ont trois attributs qui caractérise leurs mouvements : la position, la vitesse et l'accélération.

Dans un premier temps, la position est mis à jour en fonction de l'accélération et de la vitesse au temps t_{-1} suivant la formule suivante :

$p_t = p_{t-1} + v_{t-1}\Delta t + \frac{1}{2}a_{t-1}\Delta^2$

Ensuite l'accélération est mis à jour suivant la formule suivante :

$a_t = \frac{1}{2}(a_{t-1} + new\_acc)$

La valeur de new_acc est donné grâce au principe fondamentale de la dynamique :

$\sum Force = ma \Rightarrow a = \sum Force\frac{1}{m}$

Pour éviter les calculs inutile de l'inverse de la masse, celle-ci est stocké en tant qu'attribut invMass dans les objets "dynamics".

Pour permettre d'appliquer ce principe de la dynamique, notre moteur intègre le principes de force. La première force que nous appliquons à tous nos objets est la force de gravité. Les forces sont géré grâce au fichier "force.js". Chaque nouvelle force hérite d'une interface Force dont la signature est d'avoir une méthode "computeForce" renvoyant l'intensité de la force. La force de gravité est calculé par la formule $f_{grav} = m * g$ où g est l'intensité de pesanteur sur Terre : $9,81 m/s^2$.

Finalement, la vitesse est mis à jour en dernière via la formule suivante :

$v_t = v_{t-1} + a_t\Delta t^2$

Pour éviter certain problème notamment pour la résolution de collision, si la vitesse passe sous un certain seuil, elle est mise à zéro.

Nous allors maintenant expliquer les phases de détection et de résolution de collision. Ces phases sont effectué par le fichier "collisionEngine.js".

2.3 Détection de collision

La phase de détection de collision est chargé de determiner l'ensemble des objets étant en collision. Pour ce faire, on utilise une méthode brute-force en $O(n^2)$ où n est le nombre d'objet dynamique qui consiste à vérifier, pour chaque objet, si il est collision avec un des autres. On réduit la complexité en évitant de comparer deux fois le même couple. De plus, on compare uniquement les objets en états "awake" avec tous les autres.

Parlons justement de cette état, chaque objets dynamique peut être dans trois états différents, l'état awake, sleeping et static. L'état awake signifie que l'objet est en mouvement, l'état sleeping signifie que l'objet ne bouge plus et enfin l'état static signifie que l'objet n'est pas soumis à la physique mais les autres objets peuvent toujours entrer en collision avec celui-ci. L'état sleeping n'est en revanche pas utilisé dans le rendu final car nous n'avons pas trouvé de manière satisfaisante d'effectuer la transition entre l'état awake et sleeping. Seul les états awake et static sont vraiment utilisé.

Pour en revenir à la détection de collision, celle-ci est effectué grâce à l'algorithme GJK du nom de leurs inventeurs : Gilbert, Johnson, Keerthi. Il s'agit d'un algorithme permettant de détecté la collision de deux polygones convexe et uniquement convexe. Nos polygones convexe sont géré dans le dossier "shape" de nos script et l'algorithme GJK est effectué dans le fichier "shape.js".

Il s'appuie sur la différence de Minkowski. L'algorithme naif pour le calcul de collision est d'utilisé la différence de Minkowski sur tous les points des deux polygones et de vérifié si l'origine est contenue dans l'enveloppe de la différence de Minkowski.

L'algorithme GJK évite le calcul complet de la différence de Minkowski et ne l'effectue que pour un petit sous-ensemble de points des deux polygone. Le principe est d'effectuer la différence de Minkowski sur les points les plus éloigné des deux polygones. Les points les plus éloigné étant donner par la fonction de support du polygone. Une fois que l'on calcul ces points de la différence de Minkowski, on vérifie si l'origine n'est pas contenue dans le simplexe. Si elle n'y est pas alors il n'y a pas collision sinon la collision est detecté.

En sortie de cette phase de détection, on obtient tous les couples d'objet en collsion et on garde en mémoire le simplexe obtenue avec GJK car la phase de résolution de collision en aura besoin.

2.4 Résolution de collision

La phase de résolution de collision consiste à effectué les actions requises permettant de positionner les objets correctements vis-à-vis de la collision et de calculer les modifications de vitesses

correspondant à l'intensité de la collision.

Pour effectuer cela, nous utilisons un algorithme appelé EPA pour expanding polytope algorithm qui nous permet de calculer deux informations nécessaire pour la résolution de collision. Il utilise pour cela le simplexe calculé grâce à l'algorithme GJK qu'il va étendre jusqu'a obtenir les informations que nous recherchons. L'algorithme EPA est également implémenté dans le fichier "shape.js".

Ces informations sont la normal à la collision et la norme du vecteur de pénétration de la collision. La norme du vecteur de pénétration nous permet de remettre à la bonne position les polygones en collision. La normal à la collision est utilisé pour beaucoup de chose dans la résolution de collision.

2.4.1 Impulsion

L'impulsion est la phase de la résolution de collision permettant de modifié les vitesses de chacun des objets en réponse à la collision. L'impulsion est une simple modification des vecteurs vitesse de nos deux objets. Si l'on note A et B nos deux objets alors les nouvelles vitesses seront :

$\overrightarrow{v}_{a_{t+c}} = \overrightarrow{v}_{a_{t-c}} + \frac{j_a}{m_a}\overrightarrow{n}$

$\overrightarrow{v}_{b_{t+c}} = \overrightarrow{v}_{b_{t-c}} - \frac{j_b}{m_b}\overrightarrow{n}$

où $\overrightarrow{v}_{a_{t+c}}$ est la vitesse de a après la collision, $\overrightarrow{v}_{a_{t-c}}$ est la vitesse de a avant la collision, m_a est la masse de a, j_a est la modification de vitesse. Cela équivaut pour l'expression de b, la différence étant que pour b, on soustrait le facteur d'impulsion alors que pour a, on l'ajoute.

L'expression de j_a et j_b est la suivante :

$j_a = \frac{-(1+C_{r_a})(\overrightarrow{v}_{a_{t-c}} - \overrightarrow{v}_{b_{t-c}})}{(\frac{1}{m_a}+\frac{1}{m_b})} \cdot \overrightarrow{n}$

$j_b = \frac{-(1+C_{r_b})(\overrightarrow{v}_{a_{t-c}} - \overrightarrow{v}_{b_{t-c}})}{(\frac{1}{m_a}+\frac{1}{m_b})} \cdot \overrightarrow{n}$

où C_{r_a} est le coefficient de restitution de chaque objet. Plus le coefficient de restitution est proche de 0, plus l'objet rebondira et inversement.

2.4.2 Force impliqué dans une collision

Lorsqu'une collision est résolue, on ajoute la force de friction aux deux objets durant la résolution. Cela permet de diminuer l'energie cinétique des objets lors d'une collision. Cette force hérite de l'interface Force dans le fichier "force.js". L'expression du calcul de cette force est :

$F_{friction} = -\mu\overrightarrow{n}\hat{v}$

où $\hat{v}$ est le vecteur vitesse normalisé, $\overrightarrow{n}$ est la normal à la collision et μ est l'intensité de la force de friction

3 Moteur logique

Dans cette section, nous allons plutôt parler du code dans le dossier gamelogic et shape, ainsi que des levels dans le dossier ressource.
Donc nous allons d'abord parler de la gestion des Niveaux via les fichiers JSON.
En effet dans le dossier 'level' nous avons un fichier levelDescriptor.json qui donne le lien de chaque niveaux, chacun trouvable dans leur sous-dossier correspond.
De plus, ce fichier permet de sauvegarder le score qu'on a obtenue en résolvant le niveau concerné.
Il permet aussi de "signaler" au 'menu' si un niveau est accesible, ou non, pour l'utilisateur.
Bien sur ce fichier json contient le chemin des niveaux qui sont chacun d'autres fichiers json. Comme ca l'utilisateur pourra recevoir toutes les informations des niveaux via le server à n'importe qu'elle moment.
Et chaque fichier .json des niveaux contient les caractéristiques suivant :

- idlevel : Unique à chacun pour reconnaitre les niveaux qu'on appelle.
- nbBall : Donne le nombre Max de ball qu'on peut tirer dans le niveau (Ce nombre n'est pas modifier lorsqu'on joue, il sert de référence pour une variable qui décrémentera à chaque tir).

Comme ca on peut relancer le niveau avec le même nombre Max de ball (dans le cas ou il faudrait relire le json, mais ca ne vas pas être notre cas).

- target : Donne la position, point de vie, superficie de la cible que l'on veut abattre. Donne aussi l'information sur le score qu'on gagnera lorsqu'on tuera la cible.
- wall: Donne la position, mass et restitution d'un obstacle. Sa restitution est sa capacité a rebondir (proche de 0 => rebondit bien, proche de 1 c'est l'inverse). l'attribut shape permet de donner la forme de l'objet car on va analyser cette attribut pour le faire correspondre à notre classe de même nom qui dérive de la classe shape.

Maintenant il faut savoir comment on récupère les infos de ces fichiers .json.
Dans le fichier utils.js (du dossier script), il existe une fonction 'loadFromServer(url)' inspiré du cours numéro 5.

Enfin il faut parler du fichiers gameLogic.js qui possède la classe GameLogic s'occupant de tout le mode Vue du jeu.
ceete classe possède tout les informations fixe dans son constructeur (exemple, position du cannon forme du cannon, taille du sol et des limites du jeu par des murs), ainsi que la gestion élément html.
Car oui, tout les niveaux et toutes leurs informations sont affichées dans notre unique canvas d'id="backgroung_canvas" quand un niveau est appelé.
Nous possèdons aussi trois div respectivement d'id : "menu", "win", "loose".
Où le div "menu" permet l'affichage du menu avec des boutons correspondant au niveau qu'on veut appeler (tant que nous avons pas débloqué un niveau le bouton est inutilisable via la propriéte "disabled" des bouttons!!).
Le div "win" qui s'affiche quand nous avons réussi un niveau en montrant notre score et deux boutons nous permettant de revenir au Menu ou bien de passer au niveau suivant.
Enfin le div "loose" qui s'affiche quand on échoue un niveau et donne deux bouttons qui permettent de soit revenir au menu soit de recommencer le niveau échoué.
Puis, dans cette classe nous avons plusieurs fonction tel que "updateGame()" qui réaffiche tout le canvas si nous sommes dans un niveua (vérifier par l'attribut isInGame). Ceci permet d'afficher les modifications du nieveau et la suppression d'objet lorsqu'elle n'ont plus de point de vie, elle permet aussi le controle de la victoire ou de la défaite du niveau.
Nous avons aussi la fonciton "clearContext()" qui permet de repositionner la caméra à son endroit initial.
Ensuite, nous avons la fonction "enterlevel()" qui rend l'attribut isInGame a true et met tout les div en display à none (pour ne pas être affiché à l'écran) sauf le canvas. Il redonne aussi les propriété du canon et des murs limites au controleur, car celle-ci sont retiré lorsqu'on appelle la fonction "leftGame()".
Donc nous avons une fonction "leftGame()", qui sauvegarde le score que nous avons fait dans le niveau, mais il retire tous les élément qui était présent en vidant l'attribut gameObject par un tableau vide et retire aussi le mode controleur pour enfin retirer l'affichage des div win et loose et du canvas.
De plus, nous avons toutes les fonctions permettant la créations des obstacles et cibles dans le canvas (qui sont données lors de la lecture du fichier .json), avec "createRectWall(descJsonWall, id)", "createTriangleWall(descJsonWall, id)", "createWall(descJsonWall, id)", "createTarget(targetdescription, id)".

Viens ensuite la fonction "newMenuButton(specificLevelDesc)" qui créer les boutons pour le div menu, où specificLevelDesc est le tableau de donnés du fichier levelDescriptor.json. Ainsi la première fois qu'on click sur le bouton qu'on veut, on fait un appel au serveur pour récupérer le fichier du niveau correspondant via la fonction "loadFromServer(url)" de utils.js et on a le bon chemin via le tableau obtenu par levelDescriptor.json. Lors de la lecture de ce fichier level"X".json on stock le JSON.parse dans un autre tableau nommé tabLevel, qui contiendra tout les JSON.parse de tout les niveaux la première fois qu'il seront appelé! Ainsi on évite de redemander au serveur tout le fichier. On fait qu'une fois la création des boutons car pour rendre accessible un bouton d'un niveau ultérieure nous devons juste modifier l'attribut "disabled" du button. Et pour vérifier

si nous avons déjà charger le niveau (pour éviter l'appel du server) nous avons juste à vérifier si le nom du niveau est déjà présent dans tabLevel.

La fonction "initLevelMenu()" fait l'appel a newMenuButton avec son levelDescriptor.json. Il permet la création du Menu au tout début puis n'est pas réutilisé.
A contrario de "backtoMenu()" qui permet de réafficher le div "menu" et est appelé à chaque fin de niveau.
Dans le même style, nous avons "loadTheLevel(number,mySelf)" qui appels et charge les attribut de tabLevel[number].

Puis nous avons deux fonctions pour générer la fenêtre de victoire ou défaite avec la fonction "youWon(win)" et "youLoose(loose)" qui ont comme argument le div correspondant. "youWon(win)" génère deux boutons "BacktoMenu" et "NextLevel" dont l'un nous remets au Menu et le second nous fait passer au niveau suivant (soit il affiche le div "Win"). "youLoose(loose)" génère deux boutons "BacktoMenu" et "Retry" où "BacktoMenu" est similaire a celui de "youLoose()" (juste il ne donne pas accès au niveau suivant en laissant le "disabled" sur le button).

4 Conclusion

Enfin, nous avons créé un server qui permet la connexion entre nos descriptions de niveaux et le site web via un node .js judicieusement nommé 'server.js'. Ainsi l'utilisateur n'a plus qu'à lancer la commande 'node server/server.js' pour activer l'url et aller sur son 'localhost:8000/' pour voir apparaitre le jeu sur son navigateur web préféré. Une fois cela fait, l'utilisateur aura un menu avec le choix des niveaux qui se débloqueront au fur et à mesure des niveaux réussis. Pour relancer le jeu depuis le début, celui-ci peut rafraichir la page (Ctrl+R). Pour que l'utilisateur sache quelle touche de son clavier fait quelle action, il pourra lire le README.md
Voici maintenant la liste des bugs encore actifs à ce jour:

- Ctrl+S : Si vous faites Ctrl+S alors que vous êtes dans le jeu, et bien celui-ci va continuer de penser que la touche S est enfoncée et donc la camera va descendre indéfinniment.
- Next Level : Dès fois, lorsque nous appuyons sur le boutton Next Level (une fois le niveau actuel réussi), celui-ci fait aussi réussir le niveau suivant sans même l'avoir joué. Et quand on relance l'application, dès fois il n'y a plus ce soucis. Nous pensons que le problème vient lors de l'appel de leftGame() et de enterLevel().

Merci d'avoir lu ce rapport jusqu'au bout!

5 Références:

- Pour l'aspect détection et résolution des collisions:
  http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/
- Pour les lois de mouvement:
  https://perso.liris.cnrs.fr/nicolas.pronost/UUCourses/GamePhysics/lectures/