université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

23/01/2023

Langages dynamiques
Master 1 Informatique

TP nº 3

Projet

À partir de cette séance, toutes les feuilles sont optionnelles. Vous pouvez utiliser la séance pour concevoir et programmer votre projet. Des feuilles seront données de temps à autre pour vous inspirer un peu, c'est le cas de cette feuille.

Moteur Physique

Le but un petit moteur physique paramétrable et au comportement réaliste. Il fonctione en utilisant le principe fondamental de la dynamique ($\Sigma\vec{f} = m \cdot \vec{a}$) et utilise quelques notions de calcul vectoriel basiques (mais judicieusement appliquées). Il n'est pas nécessaire de comprendre la physique ou les maths sous-jacentes pour écrire le code. Un tel moteur pourra ensuite être utilisé dans un jeu de type « destruction de cible » (similaire à AngryBirds) ou dans un jeu de plateforme plus classique.

L'archive TP3 disponible sur la page du cours contient le code corrigé conçu lors de la séance 2, et légèrement ré-organisé/simplifié :

- répertoire `src/` les sources du projet
- répertoire `src/systems` les systèmes (au sens de l'ECS). Pour l'instant uniquement un fichier `draw.ml` contenant le système de dessin, un fichier `collision.ml` et un fichier `move.ml` pour les déplacements.
- répertoire `src/core` : contenant des modules utilitaires. Y sont placés un fichier `vect.ml` représentant un vecteur en 2D et `rect.ml` représentant un rectangle de taille entière.
- répertoire `src/components` : un répertoire où sont stockés les objets du jeux (les entités). On y a ajouté un fichier `block.ml` qui construit un rectangle coloré, comme lors du TP 2. La fonction `Block.create` et les fonctions associées sont appellée depuis `game.ml` et `input.ml`.

Questions

1. Lancer le jeu. On obtiendra initialement une fenêtre comme celle-ci :

2. presser la touche `N` on constate que des blocs noirs avec une vitesse initiale sont créés, mais que les collisions ne sont pas encore detectées.
3. lire le code pour comprendre exactement le cheminement logique entre la pression sur la touche `N` et l'apparition des blocks
4. lire le fichier `component_defs.ml` constater que de nouvelles interfaces sont apprues ou ont été modifiées :
   - `collidable` fait maintenant intervenir la masse
   - `physics` représente des objets ayant une masse et des forces soumises. Cette interface sera utile par la suite mais n'est pas utilisée pour ce TP.

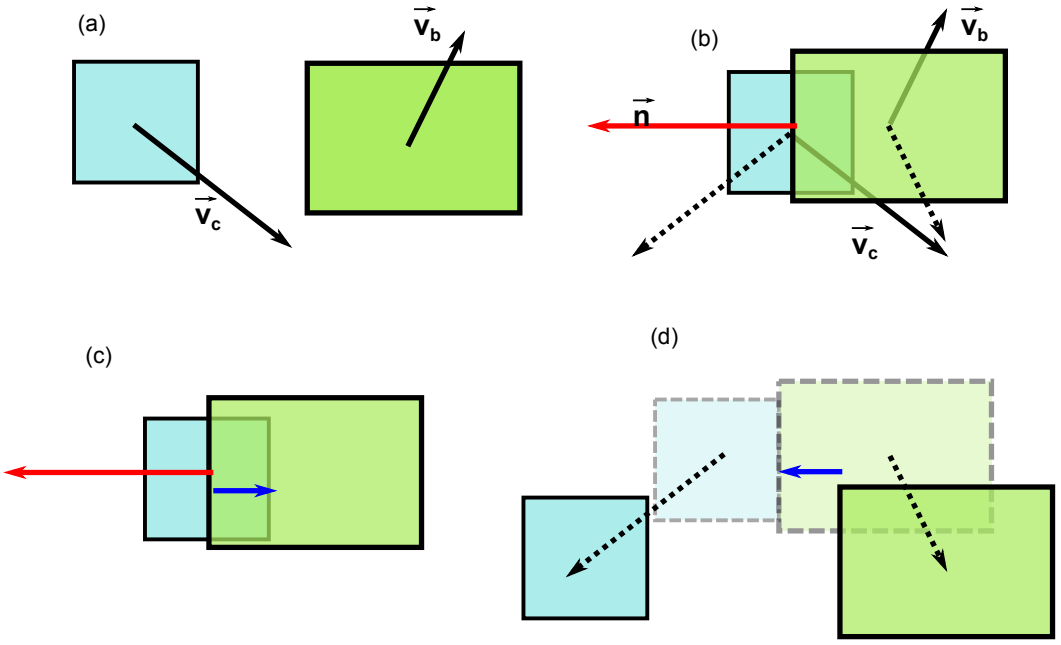

Figure 1 – Illustration de la résolution de collision entre l'objet `c` (bleu clair) et l'objet `b` (en vert).

Gestion des collisions

La partie la plus complexe de cette feuille est la gestion des collisions. Elle repose sur plusieurs concepts qu'on explique maintenant.

(a) Ajouter au fichier `rect.ml` une fonction `mdiff : Vector.t -> t -> Vector.t -> t -> Vector.t * t` telle que `mdiff p1 r1 p2 r2` calcule la *soustraction de Minkowksi* entre le rectangle `r1` positionné en `p1` et le rectangle `r2` positionné en `p2`. Cette différence est un *nouveau* rectangle défini de la manière suivante. Si on appelle x_1, y_1, w_1, h_1 les coordonnées et dimensions du rectangle `p1, r1` et x_2, y_2, w_2, h_2 celles de `p2, r2`, la soustraction de Minkowski est un rectangle défini par :

$$\begin{aligned} x &= x_2 - x_1 - w_1 \\ y &= y_2 - y_1 - h_1 \\ w &= w_2 + w_1 \\ h &= h_2 + h_1 \end{aligned}$$

(b) Ajouter une fonction `has_origin: Vector.t -> t -> bool` `true` si et seulement si le point $(0, 0)$ est contenu dans le rectangle donné en paramètre par sa position et ses dimensions.

Nous représentons les objets par leur AABB (*axis-aligned bounding box*), *i.e.* des rectangles dont les côtés sont parallèles aux axes (x, y). Cette technique est utilisée dans de nombreux jeux (et peut se généraliser en 3D avec des parallépipèdes. Évidemment on peut dessiner un objet de forme *non-rectangulaire* dans la boîte, une voiture, un personnage, *etc.*, mais on utilisera sa boîte pour détecter des collisions). Tout le code doit être placé dans la fonction update du système `Collision`. On applique l'algorithme suivant :

(a) Étant donnés deux objets e_1 et e_2 est-ce qu'au moins l'un des deux est de masse finie ? (les murs sont de masse infinie et ne rentrent donc pas en collision entre eux).

(b) Est-ce que les deux rectangles associés aux deux objets s'intersectent ? Si ce n'est pas le cas, pas de collision, on peut s'arrêter.

(c) Si les rectangles s'intersectent, on trouve *le vecteur normal* au point d'intersection, $\vec{\mathbf{n}}$. Dans notre modèle simplifié, ce vecteur est perpendiculaire aux faces qui se rencontrent (voir figure 1 (b)). C'est par rapport à ce vecteur que sont calculées les nouvelles vitesses qui représentent le « rebond » des objets (en pointillé sur la figure).

(d) On ajuste la position des objets. Cette partie est nécessaire et cause de bug graphiques si elle est mal implémentée (les objets « coulent » les uns dans les autres). En effet, s'il y a collision, les objets sont partiellement superposés. Supposons qu'ils ont une vitesse de rebond très faible (ou

nulle, par exemple dans le cas d'un mur), alors à l'itération suivante, ils ne se seront pas assez déplacés pour se séparer et seront toujours en collision. Cependant, les vecteurs vitesse des objets sont maintenant orientés de manière à séparer les objets. On a donc deux objets qui se s'éloignent mais qui sont l'un dans l'autre. L'algorithme de collision peut ne pas prévoir ce cas, ou alors considérer que l'un des objets est à l'interieur de l'autre et qu'il se cogne en essayant de sortir, et va donc le faire rebondir dans la mauvaise direction …. On doit donc déplacer les objets d'un montant équivalent au vecteur de pénetration (figure 1 (c), en bleu), dans la direction $\vec{\mathbf{n}}$ pour l'objet c et dans la direction $-\vec{\mathbf{n}}$ pour l'objet b.

(e) Une fois calculé $\vec{\mathbf{n}}$, on applique les formules de la mécanique du point pour calculer les vecteurs de rebonds. Ces calculs font intervenir les vitesses initiales, les masses des objets (un objet léger aura du mal a « pousser » un objet plus lourd, et en particulier il ne poussera pas du tout un objet de masse infinie comme un mur).

La soustraction de Minksowski $\mathtt{s} = \mathtt{b}_2 \ominus \mathtt{b}_1$ entre les deux boîtes possède des propriétés particulièrement intéressantes.

— Si $(0, 0)$ est dans le rectangle s, alors b et c sont en collision
— La plus petite distance entre $(0, 0)$ et un bord de s donne exactement le vecteur de pénétration (figure 2).

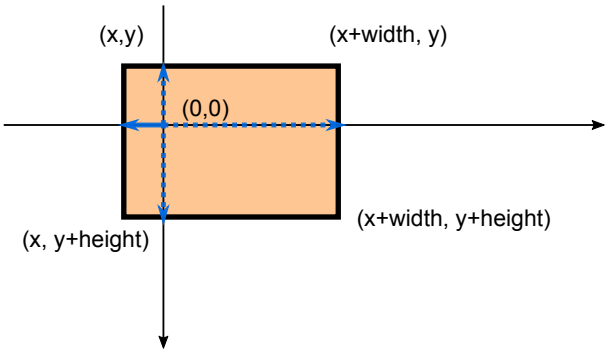

Figure 2 – Illustration de la différence s de Minkowski entre les deux rectangles de la figure 1

Algorithme détaillé de calcul des collisions On suppose que cet algorithme est implémenté dans un système `Collision` dans un fichier `collision.ml`. Pour chaque paire d'entités (e_1, e_2) ajoutées à ce système :

— Vérifier qu'au moins l'une de leur deux masse est finie
— Calculer $\mathtt{s} = \mathtt{b}_2 \ominus \mathtt{b}_1$ (où $\mathtt{b}_i$ est la boîte de l'entité e_i. Attention, une boîte est la paire d'une position et d'un rectangle, dans le code OCaml).
— Si s ne contient pas l'origine (utiliser `mdiff` et `has_origin`), alors ne rien faire.
— Sinon, utilise `Rect.penetration_vector` pour calculer parmi les 4 vecteurs bleus de la figure 2 celui qui à la norme la plus petite (lire le code). On appelle $\vec{\mathbf{n}}$ ce vecteur.
— calculer le rapport des vitesses entre e_1 et e_2 :

$$N_1 = \frac{|\vec{v}_1|}{|\vec{v}_1|+|\vec{v}_2|}$$

$$N_2 = \frac{|\vec{v}_2|}{|\vec{v}_1|+|\vec{v}_2|}$$

Cela permet de « replacer » les objets proportionnellement à leur vitesse (dans le cas où l'un des objets est un mur de vitesse nulle, on déplace complètement l'autre objet, si les deux objets ont la même vitesse, on déplace chacun de la moitié de $\vec{\mathbf{n}}$, …). Si $|\vec{v}_\mathtt{b}| = |\vec{v}_\mathtt{c}| = 0$, alors : On déplace e_1 de $N_1 \times \vec{\mathbf{n}}$ et e_2 de $-N_2 \times \vec{\mathbf{n}}$

— On normalise $\vec{\mathbf{n}}$ (on le rend de taille 1)
— On calcule la vitesse relative $\vec{v} = \vec{v}_1 - \vec{v}_2$
— On calcule l'impulsion j :

$$j = \frac{-(1+e) \times \vec{v} \cdot \vec{\mathbf{n}}}{\frac{1}{M_1} + \frac{1}{M_2}}$$

où M_i représente la masse de l'objet i et « $\cdot$ » représente le produit scalaire de deux vecteurs. Ici $e = 1$ (élasticité parfaite). Si on remplace par $e = 0$, les objets absorbent intégralement les chocs et ne rebondissent pas (et une valeur entre 0 et 1 absorbera plus ou moins les chocs créant des rebonds plus ou moins forts). En pratique e n'est pas une constante du système mais est le *coefficient de restitution* et dépend des matériaux des deux objets (par ex : acier/acier $= 19/20$, bois/bois $= 1/2$, …).

— On clacule les nouvelles vitesses et on les mets à jour :

$$\begin{aligned} \vec{v}_1 &= \vec{v}_1 + \vec{\mathbf{n}} \times \frac{j}{M_1} \\ \vec{v}_2 &= \vec{v}_2 - \vec{\mathbf{n}} \times \frac{j}{M_2} \end{aligned}$$

Ces deux quantités sont les forces auxquelles sont soumises c et b lors du choc.

On peut ensuite s'amuser à faire varier les masses, et divers coefficients, à ajouter des objets de tailles et masses différentes sur la scène, à modifier le style graphique (en fonction de la vitesse par exemple). On remarque aussi que si les objets vont trop vites, ils passent à travers les murs.

On pourra enfin essayer d'utiliser `resolver` pour détecter que deux objets sont en collision et faire quelque chose.

Deux choses restent à mettre en place (ils feront l'objet d'une dernière fiche) :

— utiliser le temps passé (le fameux argument dt) passé à tous les systèmes mais pour l'instant ignoré, afin d'obtenir une animation plus fluide et une physique un peu plus réaliste

— ajouter la gravité et contrer les effets des approximations (objets qui se traversent, collision trop rapides et donc non détectées)