université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY
Langages dynamiques
Master 1 Informatique

23/01/2023

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 :

- 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.
- répertoire `src/utils` : 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/entities` : 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. La fonction `Block.make` est appellée depuis `game.ml`.

Questions

1. Modifier le fichier `game.ml` pour faire en sorte de créer quatre blocs qui serviront de murs et délimiterons l'écran. Les murs horizontaux sont bleu et les murs verticaux sont noirs, fait 40 pixels dans sa plus petite dimension et occupe toute la longueur ou toute la largeur de l'écran, un mur sur chaque bord. Attention, les blocs ne doivent pas se chevaucher. Le code de création des quatres murs sera dans une fonction `init_walls : unit -> unit` (on pensera à commenter le code lié au carré multicolore, et on le supprimera petit à petit). On obtiendra initialement une fenêtre comme celle-ci :

2. Créer (dans `component_defs.ml`) de nouveaux composants :
   - `mass` : un flottant représentant la masse d'un objet
   - `velocity` : un vecteur représentant la vitesse d'un objet
   - `sum_forces` : un vecteur représentant la somme des forces auxquelles un objet est soumis

Modifier la fonction `Block.make` pour prendre en argument la masse du bloc. Ajouter au bloc des composants `mass`, `velocity` et `sum_forces`, ces deux derniers initialisés au vecteur nul. Modifier enfin `Game.init_walls` pour créer des murs de masse infinie (`infinity` représente l'infini positif en nombre flottants).

3. Ajouter maintenant un nouveau système, `move.ml`. On s'inspirera pour cela de `draw.ml`. Ce système se charge de déplacer les entités associées en fonction de leur vecteur vitesse. Les entités associées doivent donc avoir un type demandant les deux propriétés `position` et `velocity`. Pour déplacer un objet, on procède de la façon suivante. On pose `dt = 1000. /. 60.` (le temps d'une frame). La nouvelle position est l'ancienne position à laquelle on ajoute $dt \times \overrightarrow{v}$ où $\overrightarrow{v}$ est le vecteur vitesse de l'objet. Ne pas oublier d'enregistrer ce système dans `system_defs.ml` et d'appeler sa fonction de mise à jour dans le programme principal.
4. Dans `Block.make`, enregistrer le block créé auprès du système `Move`. Créer ensuite dans le programme principal un bloc carré rouge de masse 10.0, rouge, puis initialiser sa vitesse au vecteur $(0.25, 0.25)$. Constater que le bloc se déplace (mais qu'il n'est pas arrêté par les murs).
5. Ajouter maintenant un nouveau système, `forces.ml`. Ce dernier à pour rôle de transformer toutes les forces auxquelles un objet est soumis en accélération, puis l'accélération en vitesse. C'est dans ce système qu'on pourra aussi appliquer à tous les objets des forces perpétuellement présentes (la gravité, les frottements, ...). Le sytème requiert la présence des propriétés `sum_forces`, `mass` et `velocity`. Comme pour le système `Move`, il faut d'abord définir `dt = 1000.0 /. 60.0`. Puis, pour chaque entité enregistrée :
   — récupérer la masse de l'objet. Si cette dernière est infinie, ne rien faire. Cela nous permet de traiter simplement les objets perpétuellement fixes, comme les blocks qui représenterons le sol, les murs, les plateformes. On pourra utilise `Float.is_finite` pour tester la finitude d'un flottant.
   — récupérer la valeur du vecteur `sum_forces` le diviser par la masse. En effet, l'équation fondamentale de la dynamique nous dit que $\Sigma \overrightarrow{f} = m\overrightarrow{a}$ et donc $\overrightarrow{a} = \dfrac{\Sigma \overrightarrow{f}}{m}$. On note qu'à ce stade, diviser par une masse infinie donnerait 0 et donc pas de modification d'accélération ni de vitesse.
   — la multiplier par `dt` (l'accélération multipliée par le temps donne la vitesse)
   — ajouter ce nouveau vecteur à la vitesse de l'entité
   — réinitialiser `sum_forces` au vecteur nul

   Une fois fait, enregistrer dans `Block.make` les blocs auprès du système `Forces`. Remplacer maintenant la vitesse initiale par un vecteur force de valeur $(0.25, 0.25)$. Constater que l'objet se déplace de la même façon. En effet, plutôt que de commencer avec une vitesse constante, il commence avec une impulsion initiale, qui lui donne une accélération linéaire (sur la première frame), celle ci se traduisant en une vitesse constante.

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.

6. 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 :

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

7. 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

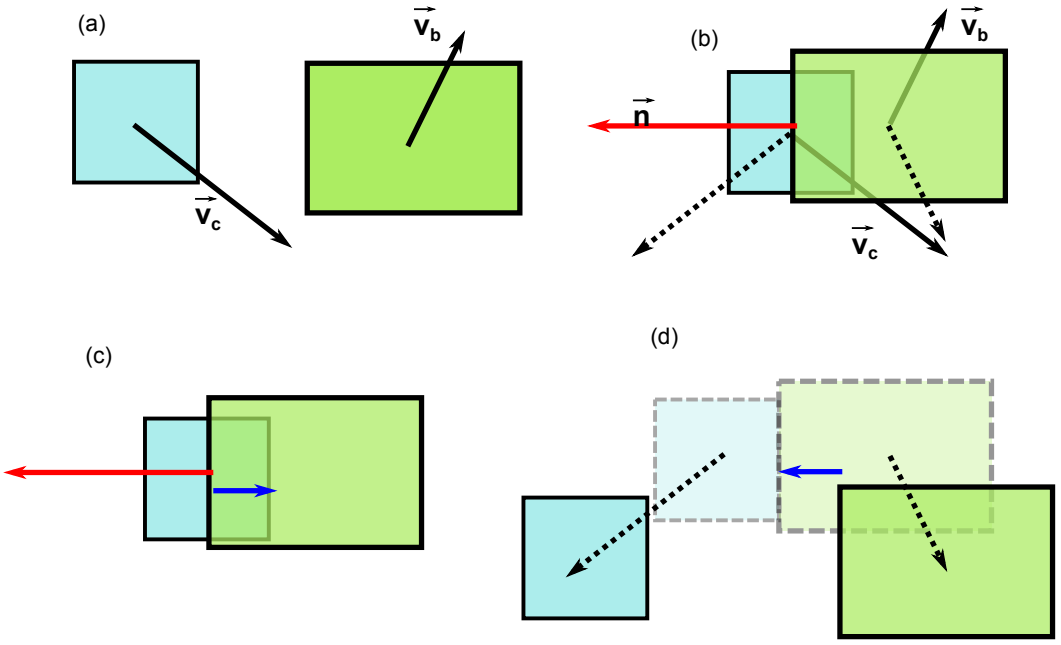

Figure 1 – Illustration de la résolution de collision entre l'objet c (bleu clair) et l'objet b (en vert).

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 méthode `.collision(b)` de la classe `Body`. Cette dernière renvoie `null` si l'objet courant (nommé c dans la suite) et l'objet cible b n'ont pas de collision et sinon renvoie leur nouvelle vitesse. La figure 1 illustre le phénomène. On applique l'algorithme suivant :

1. Est-ce que les deux rectangles s'intersectent ? Si ce n'est pas le cas, pas de collision, on peut s'arrêter.
2. 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).
3. 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.
4. 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 $\mathtt{s} = \mathtt{b} \ominus \mathtt{c}$ 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).

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 $(,)$ ajoutées à ce système :

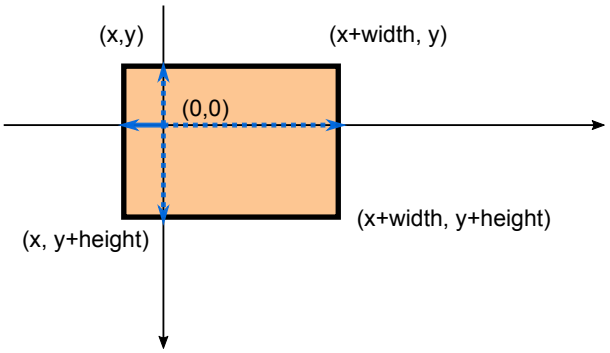

Figure 2 – Illustration de la différence s de Minkowski entre c et b

— Calculer $\mathsf{s} = \mathsf{b} \ominus \mathsf{c}$
— Si s ne contient pas l'origine (utiliser `mdiff` et `has_origin`), alors ne rien faire.
— Sinon, calculer les 4 vecteurs bleus de la figure 2 et garder celui qui à la norme la plus petite. On appelle $\vec{\mathbf{n}}$ ce vecteur.
— calculer le rapport des vitesses entre b et c :

$$N_\mathsf{c} = \frac{|\vec{v}_\mathsf{c}|}{|\vec{v}_\mathsf{c}|+|\vec{v}_\mathsf{b}|}$$
$$N_\mathsf{b} = \frac{|\vec{v}_\mathsf{b}|}{|\vec{v}_\mathsf{c}|+|\vec{v}_\mathsf{b}|}$$

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 $|\vec{v}_\mathsf{b}| = |\vec{v}_\mathsf{c}| = 0$, alors :
— Si les deux objets sont de masse infinie, la fonction ne fait rien et passe à la paire suivante (on considère que deux murs ne font pas de collision)
— Si b est plus lourd que c on définit $N_\mathsf{c} = 1$ et $N_\mathsf{b} = 0$, sinon on définit $N_\mathsf{b} = 1$ et $N_\mathsf{c} = 0$. (i.e. on considère que l'objet le plus lourd des deux « éjecte » le plus léger).
On déplace c de $N_\mathsf{c} \times \vec{\mathbf{n}}$ et b de $-N_\mathsf{b} \times \vec{\mathbf{n}}$
— On normalise $\vec{\mathbf{n}}$ (on le rend de taille 1)
— On calcule la vitesse relative $\vec{v} = \vec{v}_\mathsf{c} - \vec{v}_\mathsf{b}$
— On calcule l'impulsion j :

$$j = \frac{-(1+e) \times \vec{v} \cdot \vec{\mathbf{n}}}{\frac{1}{M_\mathsf{c}} + \frac{1}{M_\mathsf{b}}}$$

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 :

$$\vec{v}_\mathsf{c} = \vec{v}_\mathsf{c} + \vec{\mathbf{n}} \times \frac{j}{M_\mathsf{c}}$$
$$\vec{v}_\mathsf{b} = \vec{v}_\mathsf{b} - \vec{\mathbf{n}} \times \frac{j}{M_\mathsf{b}}$$

Ces deux quantités sont les forces auxquelles sont soumises c et b lors du choc. On ajoute ces forces à `sum_forces` pour chacun des objets.
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. Une technique plus élaborée de collision peut être mise en œuvre (*swept* collision ou *continuous collision detection*).

8. implémenter la détection de collision comme un système puis tester

9. ajouter d’autres objets mobiles avec des conditions initiales différentes et vérifier que les collisions se font correctement