Performances du moteur de jeu

immediate

January 12, 2020

1 Moteur de jeu

Le moteur de jeu est basé sur des itérations de fonctions main_loop. Cette fonction regroupe tous les aspects du jeu de la gestion des inputs au moteur physique en passant par l'affichage des données. Il est nécessaire de donner un dt à cette fonction afin de signaler pour combien de temps on veut calculer l'itération. Ce temps est essentiellement utilisé par le moteur physique pour calculer l'inertie sur une durée donnée. Il n'y aura qu'une seule image affichée pour toute cette durée. Ceci veut dire que si dt est très petit le jeu semblera lent. En effet a un nombre fixé d'ips le moteur physique ira plus lentement que si dt est élevé. Ceci a été crucial pour notre projet car il est assez lourd et demande beaucoup de calcul vu la vitesse à laquelle se déplace le personnage. De plus nous avons utilisé python qui n'est pas connu pour sa rapidité ce qui a souvent été un problème. Ainsi pour que le jeu marche également sur les petites configurations il a fallu faire un compromis entre propreté et polyvalence du code et rapidité (par exemple fusionner des boucles for qui n'avaient rien à voir mais qui tournaient sur le même ensemble de manière indépendante, faire des approximations physique, ...). Ces points seront abordés dans la suite.

2 Calcul du dt

Comme nous avons pu le voir juste avant le calcul du dt est une chose cruciale dans les performances du jeu. En effet sur une petite configuration, il y aura moins d'ips que sur une grosse. Or on aimerait avoir un jeu se déroulant à la même vitesse qu'importe la configuration. De cette façon ça évite les bugs comme par exemple si on faisait tourner le jeu sur un supercalculateur, si on ne frenne pas la boucle tout va se dérouler trop vite pour que le joueur puisse jouer. Si on le lance sur une trop petite configuration le jeu va être très ralenti et donc injouable voire trop facile car lent donc moins spontané. Nous avons donc tout d'abord essayé de fixer le nombre d'ips (qui est le nombre de tour de boucle par seconde). Or cette tentative a échoué car en réalité même sur une grosse configuration il est très difficile d'avoir un nombre constant d'ips. Très régulièrement il y a des baisses

et à d'autres moments on est obligé de beaucoup freiner le moteur. En fait pour avoir des performances optimales il faudrait que dt vale exactement le temps que met la boucle à s'executer et ce avant qu'elle s'execute. Ceci est bien entendu impossible à calculer en pratique. On va alors faire une approximation qui a plutôt bien marché en pratique : le temps que met la boucle a s'executer est similaire à celui qu'elle a mis pour s'éxécuter la fois d'avant. Bien entendu cette approximation est fausse car elle suppose une certaine continuité à l'échelle de nos dt (de l'ordre de 1ms en général). C'est cependant la méthode qui a donné de meilleurs résultats pour le moment.

3 Main Loop

Une fois que l'on a calculé dt il reste à faire tout le travail de la main_loop. Cette fonction va en appeler 8 autres gérant chacune un aspect du jeu :

- Animation de mort
- Calcul des objets à traiter par le moteur physique
- Calcul des controllers (inputs)
- Moteur physique
- Calcul de la position de la camera
- Affichage de la camera
- Calcul du score
- Calcul de Win/Lose

Chacun de ces points va être discuté et analysé dans la suite.

3.1 Animation de mort

Ceci est sûrement la partie la plus simple des 8. Quand le joueur a perdu, le jeu ne s'arrete pas tout de suite. Le moteur physique reprend le dessus sur la position X du joueur (voir partie 5) et le fait se balancer comme un objet physique classique. De cette façon le score apparait un peu plus tard et laisse le temps au joueur de comprendre ce qui l'a tué. Pour cela on utilise un attribut GameLevel.lost. S'il est mis à True cela veut dire que le joueur a perdu (il peut toujours gagner un peu de points pas inertie, ça fait partie du jeu voire même gagner ce qui peut donner place à des niveau assez amusants où il faut jouer sur la physique du corps du joueur pour se propulser. Le temps avant l'apparition de l'ecran des scores est stocké dans GameLevel.countdown en secondes. Quand le jeu est terminé il renvoie une exception EndGame avec les arguments (transformés en attributs) reflétant si c'est une victoire ou une défaite. Cette fonction est donc très peu coûteuse : O(1).

3.2 Calcul des objets à traiter par le moteur physique

Le moteur physique fait principalement des comparaisons 2 à 2 des objets pour voir s'ils sont en collision et si oui comment les en sortir. Or ceci à un coût en $O(n^2)$ avec n le nombre d'objets dans le niveau. Ceci devient très vite très couteux sur de gros niveaux. L'astuce utilisée ici est de ne conserver qu'une petite liste d'objets utiles mise à jour dynamiquement à chaque itération. Les objets utiles sont les objets dans le champs de la camera sauf exception. En effet dès qu'un objet considéré comme utile sort du champs de la camera par rapport à x (s'il est au dessus ou en dessous de la camera on le garde) on appelle sa méthode stase. Si elle renvoie 0 alors l'objet est retiré de la liste. Ceci permet par exemple de garder des objets qui pourraient revenir dans le champs de la camera plus tard comme par exemple des projectiles téléguidés, des monstres volants, ... Ainsi on ne travaille que avec un nombre réduit d'objets dans le moteur de jeu - bien entendu cela fait de grosses approximations nottement à coté des bords où il pourrait y avoir un objet proche mais hors du champs et du coup les collisions ne sont pas calculées correctement. Cependant ceci permet d'avoir une complexité en $O(m^2)$ avec m le nombre d'objet dans un carré de taille *constante* dans le niveau. On peut considérer que c'est constant dans une certaine mesure. En effet si on n'avait que des plateformes uniformément réparties comme c'est presque toujours le cas on pourrait faire cette approximation sans problème. Cependant dans le cas de projectiles qui orbitent autour du joueur comme dans le cas des shields c'est tout de suite moins évident. On va donc le modéliser comme ceci : $O((s+p)^2)$ avec s le nombre moyen d'objets statiques (plateformes, objets récupérables, ...) et p le nombre d'objets movants comme des projectiles. Habituellement m est faible donc c'est de l'ordre de p^2 ce qui reste plutôt cher lors de l'utilisation de gravitational shields par exemple (shield de projectiles ayant eux même un shield d'autres projectiles). Avant de lancer le jeu il y a une phase de précalculs qui vise à trier la liste des objets données pour en faire une file de x croissants (O(nlogn) avec n le nombre total d'objets). Une fois ce trie, à chaque itération on va chercher à ajouter le premier objet de la file aux objets actifs. Si c'est possible on continue jusqu'à ce qu'on ait un objet en dehors de la camera. Comme on suppose que les objets sont uniformément répartis sur le niveau qui est assez large on en ajoutera que peu à chaque fois et le même nombre sera enlevé de la liste avec ce qu'on a vu juste avant.

3.3 Calcul des controllers

Un controller est un objet lié à un ControlableNode chargé de le manipuler un peu comme une marionette. On peut le voir comme une petite IA pour les controllers automatiques. Tous les controllers sont automatiques sauf celui du joueur qui est controllé par des touches du clavier. Bien entendu on peut ajouter plus de controllers liés au clavier si besoin en permettant au joueur de jouer 2 personnages en même temps par exemple voire d'effectuer des commandes précises pour controller un drone capable d'éliminer des monstres par exemples. Ainsi chaque controller prend independament de s'il en a besoin ou pas, chaque

event pygame. La complexité est donc en O(e*m) avec e le nombre d'events et m le nombre d'objets utiles. Evidemment ce n'est pas très optimisé car la plupart des controllers vont jeter ces arguments. On pourrait ainsi avoir une complexité en O(m+e) en ne donnant aucun event aux controllers IA et seulement les events au seul controller joueur (on suppose qu'il y en a un nombre faible si ce n'est pas 1 car il devient vite difficile de jouer plus de 2 personnages à la fois). Or la boucle des controllers ne posait pas de problème particulier en terme de complexité en pratique (souvent de l'ordre de quelques ms pour les 2 boucles imbriquées. Ici on pouvait donc se permettre de rester général. En effet il pourrait être très interessant d'avoir par exemple des comportements de monstres basés sur les inputs du joueur : par exemple un monstre qui tir en l'air quand le joueur saute. Il reste donc toujours interessant de donner les inputs du joueur aux IA même si une partie d'entre elles ne les utilise pas.

3.4 Moteur physique

C'est la partie la plus technique. Il faut savoir que nous tournons actuellement sur la physique 5.0. En effet il y a eu 4 moteurs physiques antérieurs à celui ci. Tous géraient la physique à des niveaux différents et étaient trop lents pour les petites configurations. Ainsi cette version est la plus simples des 5 et vise a marcher sur des petites configuratons. *(*Cependant) nous ne recommandons pas de jouer au jeu sur de trop petite configurations. En effet vous aurez un dt élevé donc peu d'ips et vous ne pourrez pas apprécier le jeu à sa juste valeur. D'autant plus que certains niveaux demandent un certain niveau de précision dans les actions vous serez fortement désavantagés avec peu d'ips. Comme pour un incendi nous avons des dispositifs anti feu pour éviter les bugs mais s'ils sont utilisés l'experience de jeu sera très affaiblis et désagréable.

Le moteur physique est séparé en plusieurs fichiers : les classes HitBox et Rect qui font les calculs précis de hitbox et de collisions et enfin la méthode GameLevel.physics_step qui supervise le tout. On va utiliser une approche top-down pour cette analyse.

La fonction GameLevel.physics_step effectue une double boucle for sur les objets utiles et calcule la vitesse associée au dt donné et la nouvelle position du premier objet. On part donc de l'accélération obtenue à partir des forces appliquées sur l'objet pour remonter à la translation associée. Si cet objet est le joueur on force sa position en x à valoir la valeur donnée par le niveau (pour gérer le tempo de manière plus précise). Bien entendu on pourrait vouloir laisser le moteur physique gérer en donnant la vitesse associée à ce déplacement cependant dt est un flottant donc la position obtenue suite à ça diffère d'un epsilon. Ces epsilon allant s'accumuler ce n'est pas une très bonne idée de procéder comme ça d'autant plus qu'il est beaucoup plus simple de forcer la position. On tronque de même la vitesse en x. On effectue évidement ces opérations après avoir calculé la position du joueur suggéré par le moteur physique car on gardera la position qui n'est pas calculable autrement que avec les vraies valeurs. Une fois qu'on a les deux objets et calculé la nouvelle position du premier, on regarde s'ils sont en collision. Pour cela on dispose de 2 hitbox

: la rigid_hitbox et la soft. La rigid gère les comportements de collision entre rigid body et la soft n'est que là pour indiquer qu'il y a une collision. Ainsi si les 2 objets ont une soft hit box (sachant que avec une rigid hit box implique avoir une soft), que ce ne sont pas les mêmes objets et qu'ils sont en collision alors on appelle les fonctions collide de chaque objet. Ces fonctions vont par exemple retirer des points de vie au joueur s'il entre en contact avec un monstre, ... Par convention chacun gère ses propres problèmes : le joueur perd des pv s'il entre en collision avec un monstre. Le monstre meurt s'il entre en collision avec le joueur. Ces fonctions sont généralement en O(1) car ce ne sont que des asignations de variables ou des petites opérations arithmétiques. Il est également donné les faces entrées en collision (très utile pour les intéractions entre le joueur et les plateformes). Enfin si on a eu collision soft on regarde si on a collision rigid. Si oui on applique la réaction solide pour sortir de la collision (on vera cela plus en détails dans la suite). En bref cet algorithme opère en O($m^2 * (2 * CHECK\_COLLISION + APPLY\_REACTION) + m * (COMPUTE_SPEED + MOVE)$).

Regardons plus en détails la première partie de la formule. Tout objet qui entre en collision a une hitbox (ou deux si c'est un rigid body). Les hitbox sont rectangulaires et en tranlation les unes des autres pour des raisons d'efficacité (la version 4.0 gérait les rotations mais ne marchait pas bien sur les petites configurations). Ainsi pour calculer si on est en collision on calculer le rectangle intersection (O(1)), s'il est vide il n'y a pas collision sinon il y en a une. De plus on a besoin des cotés qui entrent en collision. Pour cela on réutilise le rectangle intersection et on regarde s'il est plus large que long $\rightarrow$ collision top ou bottom ou si c'est plus long que large $\rightarrow$ collision left ou right. On décide entre les 2 en fonction de la vitesse de l'objet qui vient d'entrer en collision. Si on est dans un cas top ou bottom et que sa vitesse selon y est négative c'est top, sinon c'est bottom. Ce n'est pas possible que ce soit 0 car on suppose que a la fin de GameLevel.physics_step tous les conflits sont résolus. En effets tous ceux de début de physics_step le sont car ils sont traités 2 à 2. Les seuls possibles de rester sont ceux qui ont été créés suite à la résolution d'un autre. Comme on veut éviter à tout prix d'avoir des problèmes de convergence (très mauvais pour les performances) on ne vérifier pas ces collisions collatérales. En effet elles n'arrivent que dans des environnements très encombrés ce qui n'arrivera jamais dans un niveau (tous les rigid body sont assez espacés les uns des autres). De cette façon le seul moyen pour que deux corps soient en collision rigid est que le premier vienne de bouger et ce mouvement induit une collision. De cette façon l'objet 1 a une vitesse non nulle dans le sens de la collision. Ainsi tout se fait donc en O(1). Du coté de la suppression de la collision on recalcule le rectangle d'intersection (ce n'est pas très couteux donc on peut se permettre de ne pas le transmettre en argument). Si le mouvement a été fait selon x alors le mouvement pour se sortir la collision est selon x dans le sens inverse et sur une distance égale à la largeur du rectangle d'intersection (+ un epsilon car on travaille avec des flottants et il faut être certain d'avoir résolu le conflit). Cette étape se fait donc encore en O(1)

La partie droite de la formule est plus simple. Pour calculer la vitesse après un temps dt on calcule les effets des forces ce qui se fait en O(f) avec f le nombre de forces (souvent

très faible donc c'est un O(1)). La plupart du temps on n'a que la gravité. Ceci calcule l'acceleration en O(1). Puis on en déduis par intégration numérique la vitesse. On fait de même avec la position à la différence qu'elle a être tronquée. En effet on ne veut pas avoir des translations trop grandes sinon on risque en un dt de traverser un objet sans même se rendre compte qu'il y a eu collision. Pour que les collisions soient bien calculées il faut que ce déplacement maximale soit inférieur au plus petit diametre des objets. Comme les objets sont souvent plus grand que la taille 10 on va prendre cette valeur (ce qui est problématique avec les projectiles par moment qui sont souvent très petits). Le fait de tronquer n'est très pratique car ça limite la vitesse maximale (ça limite la vitesse par itération donc par frame et puis avec les ips on obtient la vraie vitesse). Or notre jeu doit aller assez vite donc il ne faut pas mettre une translation de coupure trop faible (on a donc quelques pertes sur les projectiles avec les faibles configurations). En bref le calcul du moteur physique se fait en O(m^2).

3.5 Calcul de la position de la camera

La position de la camera est calculée à chaque frame. Elle est centrée à 3/4 sur le joueur (3/4 de la camera devant le joueur). Ceci donne sa position x. Pour sa position y elle va essayer de se fixer la plateforme en dessous du joueur avec un maximum de recherche de 100 unités de distance. Si elle ne trouve rien elle prend la valeur y du joueur. Ainsi elle permet d'avoir un point de vue stable selon y tout en suivant le joueur. Pour calculer ceci il faut trouver la plateforme en dessous du joueur (qui peut être en train de sauter) ce qui se fait en O(m) avec m le nombre d'objets actifs.

3.6 Affichage de la camera

L'affichage de la camera a subit quelques changements dernièrement à cause des problèmes de performances. En effet utiliser un blit d'une grande surface vers une autre prend beaucoup de temps (en tout cas c'est dans ce genre d'opération qu'on a le plus de problèmes). Il vaut donc mieux pour des raisons de performances (et malheureusement ce n'est pas terrible au niveau de la polyvalence) directement utiliser des blit sur la feunetre donnée par le menu. Ainsi on commence par effacer le contenu de la surface en la remplissant de noir : O($w * h$) avec w la largeur et h la hauteur de la feunetre. Puis on affiche le background : O($w * h * b$) avec b le nombre de parallax (calculer la position des parallax se fait en O(1)). Ensuite pour chaque objet on regarde s'il est dans la camera et si oui on l'affiche. Sinon on le stase et on voit si on peut l'enlever. Cet affichage se fait en O($w * h$) car très peu d'objets sont superposés et globalement cela recouvre moins que le background donc c'est borné par w*h en pratique. La partie gestion des objets se fait en O(1) de manière générale par supposition d'uniformité et de répartition. On va enfin ajouter des textes comme le score, le nom du niveau et des effets comme le poison. Tout ceci rentre dans le O($w * h$) car on les blit a des endroits où il n'y a à priori rien donc même en rajoutant ceci aux objets on

garde la même borne.

3.7 Calcul du score

Le calcul du score est simple : à chaque fois qu'on arrive à la fin d'une plateforme on gagne 1000 points. Pour cela une liste des x des fins de plateforme est précalculée et à chaque tick on regarde si on a dépassé le début de la file. Si oui on ajoute 1000 au score et on passe au suivant sinon on stop. Ici le coût à l'itération est un O(1) d'après les hypothèses de répartition et d'uniformité.

3.8 Calcul de Win/Lose

Depuis un récent patch on ne peut plus que gagner en passant à proximité d'un drapeau donc cette fonction ne calcule plus que la Lose. Pour cela elle calcule la taille du niveau (précalcul en O(n)) et son enveloppe rectangulaire. Si on dépase la hauteur y du segment du bas (si on sort de l'enveloppe) c'est perdu. Ceci se fait en O(1) après précalcul.

3.9 Conclusion

En bref en supposant une uniforme répartition des plateformes et un certain écartement entre elles, le moteur de jeu opère en une phase de précalculs en O(nlogn) et à chaque itération en O($m^2 + b * w * h$). Or comme m = s+p avec s petit par uniforme répartition. On a donc une complexité finale approximative en O($s^2 + b * w * h$). On ne peut cependant que très peu modifier w et h. Experimentalement s ¡ 10 et b ¡ 5 sont de bonnes valeurs pour des ordinateurs classiques.

4 Annexe : Coté pratique - Analyse avec cProfile

En utilisant cProfile lors du lancement du jeu, j'ai récupéré le temps d'execution total du jeu (y compris les menus) lors du lancement de différents niveaux. Il était proposé de mesurer les limites du moteur physique lorsqu'il y a beaucoup d'ennemis. Or comme les ennemis peuvent se taper entre eux il est bien plus pertinent pour avoir beaucoup d'entités chargées dans le moteur physique d'utiliser des projectiles. J'ai donc créé un bouclier qui teste les limites du moteur physique : le gravitational shield (cf Campaign/Kshan/Grave Forest). Il s'agit d'un bouclier de GravitationalBalls qui sont des projectiles ayant eux même un bouclier de projectiles simples (LaserBalls). Ainsi il y a 3 graviationalBalls ayant chacune 5 LaserBalls accrochées ce qui fait 15 entités en plus à l'écran de ce qu'on a en moyenne. Or comme nous l'avons vu la complexité en m est quadratique. J'ai donc effectué 4 mesures pendant une quinzaine de secondes : le niveau avec le gravitational shield avec et sans background ainsi qu'un autre niveau très simple avec presque que des plateformes (donc de complexité en m normale). Les fichiers sont disponibles dans le dossier mesure.

Voici les temps de calcul engendrés par les fonctions principales (cumtime per call de cProfile → temps de calcul de la fonction ainsi que le temps des fonctions appelées par cette fonction):

Analyses	high bg	control bg	high	control
Animation de mort	0.000	0.000	0.000	0.000
Calcul des objets à traiter par le moteur physique	0.000	0.000	0.000	0.000
Calcul des controllers (inputs)	0.001	0.000	0.001	0.000
Moteur physique	0.012	0.002	0.012	0.002
Calcul de la position de la camera	0.000	0.000	0.000	0.000
Affichage de la camera	0.022	0.013	0.012	0.011
Calcul du score	0.000	0.000	0.000	0.000
Calcul de Win/Lose	0.000	0.000	0.000	0.000
Total de main_loop	0.035	0.016	0.025	0.013
Ips	28	62	40	77

Ces résultats sont vraiment très propres et on voit clairement la complexité engendrée par l'ajout d'un background (ici il avait 4 parallax) et de projectiles. En effet plus il y a d'objets et plus le nombre d'ips est lent dû au moteur physique. Le background ralenti aussi mais ce background ralenti moins que le fait d'ajouter des objets. Ceci se voit d'autant plus au niveau de l'affichage de la camera. Il coûte presque aussi cher d'ajouter 15 projectiles que de mettre un background. Il est également possible de voir les détails des coûts dans les fichiers joints mais pour des raisons de temps et de révision des exams je n'ai pas le temps de rédiger un commentaire précis des coûts de chaque fonction et de faire le lien avec les choix d'implémentation.