GPA675

Laboratoire 2 – Jeu de serpent

Table des matières

Objectif principal

Ce projet a pour principal objectif d'approcher plusieurs problématiques de développement logiciel en considérant certaines structures de données, certains algorithmes et mieux comprendre leur complexité.

L'accent est mis sur le développement et l'utilisation d'une liste chaînée double.

Objectifs spécifiques

Plus spécifiquement, ce projet vise les objectifs suivants en ordre de priorité :

- analyses de complexité liées à des algorithmes spécifiques, mais aussi à ceux liés aux structures de données
- mettre en pratique le développement et l'utilisation de diverses structures données, notamment la liste chaînée double
- améliorer ses compétences avec :
  - la programmation C++
  - la programmation orientée objet
  - la lecture et l'interprétation de diagrammes UML
  - utilisation des structures de données linéaires du C++ (`std::list`, `std::vector` ou `std::array` par exemple)
- développement d'un ensemble de classes permettant la création d'un jeu de serpent flexible et modulaire mettant à profit le polymorphisme

Jeu de serpent

Introduction

Le jeu de serpent (ou « *Snake Game* ») est un genre de jeu vidéo. Le genre est devenu un classique après la création de dizaines de jeux différents s'inspirant les uns des autres, mais tous basés sur le premier du genre, « *Blockade* ».

Ce projet se veut la mise en place d'une plateforme flexible du même concept.

Concepts du jeu

Pour ce projet, plusieurs concepts sont mis de l'avant :

- l'arène est le monde dans lequel évoluent tous les éléments du jeu
- le monde est constitué de n éléments – ces derniers sont répartis en quatre familles selon deux catégories principales:
  - éléments statiques :
    - n_o obstacle
    - n_p pastilles
  - éléments dynamiques :
    - n_s serpents
    - n_r souris (en bonus)

    - $n = n_o + n_p + n_s + n_r$
- plusieurs scénarii de parties peuvent être créés à partir de cette infrastructure, les scénarii incluent tous ces éléments particuliers :
    - mise en place des conditions initiales
    - stratégie(s) d'insertion des pastilles (ou autres éléments)
    - condition de fin de partie
    - présentation du pointage

Détails sur l'arène

L'arène possède au moins ces caractéristiques :

- l'arène est à deux dimensions
- c'est un monde discrétisé formé de cellules où s'y retrouvent chaque élément du jeu
- elle est de forme rectangulaire
- sa taille est prédéterminée avant la partie et immuable pendant la partie
- la taille doit respecter ces contraintes :
    - la largeur doit être d'au moins 4 cellules et correspondre à un nombre pair
    - la hauteur doit être d'au moins 3 cellules
- elle possède *n* entités
- elle est systématiquement bordée d'obstacles au pourtour
    - une exception peut être faite si certaines cellules du contour sont des éléments de téléportation (en bonus)
- en tout temps, une cellule de l'arène ne peut avoir que 2 états :
    - vide
    - une seule entité peut occuper la cellule – chaque cellule de l'arène ne peut être occupée que par un élément à la fois
- lors de l'insertion d'une entité pendant le jeu, l'arène détermine aléatoirement une cellule disponible pour la position de l'élément à ajouter

Détails sur les entités

Les entités

Toutes les entités possèdent ces caractéristiques communes :

- elles possèdent un lien permanent vers l'arène de jeu afin d'interagir avec cette dernière
- elles ont un âge initialisé à 0 lors de l'insertion dans l'arène
- elles vieillissent à chaque pas de simulation du temps écoulé entre ces *tics*
- elles ont une durée de vie (certaines entités sont immortelles et possèdent une durée de vie ridiculement élevée)
- lorsqu'elles meurent, elles sont retirées de l'arène
- chaque entité détermine 2 comportements fondamentaux :
    - son évolution dans la partie pour chaque *tic*
    - sa représentation dans l'arène

Les entités statiques

Les entités statiques ont ces caractéristiques :

- leur position est déterminée avant l'insertion dans l'arène et reste immuable pendant toute leur vie

Les entités dynamiques

Les entités dynamiques ont ces caractéristiques :

- leur position initiale est déterminée avant l'insertion dans l'arène, la position change pendant la partie
- elles possèdent une vitesse déterminée avant l'insertion dans l'arène, la vitesse peut changer pendant la partie
- à chaque pas de simulation (*tic*) l'entité doit inéluctablement se déplacer selon sa vitesse en cellules par seconde (le déplacement n'est pas nécessairement à chaque *tic* puisque ces derniers sont à haute fréquence et que les entités dynamiques ont une vitesse en cellules / seconde)
- chaque type d'entité possède un comportement de déplacement qui lui est propre

Les obstacles

Les obstacles sont des entités statiques qui présentent ces caractéristiques :

- elles sont de la taille d'une seule cellule
- il peut y avoir n_o obstacles présents dans l'arène à tout moment ($n_o \geq 0$)
- ils peuvent être insérés à n'importe quel moment de la partie
- lorsqu'une entité dynamique se déplace sur l'obstacle, cette dernière meurt et est retirée de la simulation
- même si l'obstacle est généralement présent durant toute la durée de la partie, sa durée de vie peut être limitée

De plus, l'arène possède systématiquement quatre murs constitués d'obstacles qui forment les bords.

Les pastilles

Les pastilles sont des entités statiques. Elles sont des items qui peuvent être consommés par les entités dynamiques. Elles présentent ces caractéristiques :

- elles sont de la taille d'une seule cellule
- il peut y avoir n_p pastilles présentes dans l'arène ($n_p \geq 0$)
- elles peuvent être insérées à n'importe quel moment de la partie
- une pastille, lorsque consommée, applique un effet sur l'entité dynamique concernée :
    - l'effet de la pastille est paramétrable en amplitude
    - suivant le type d'effet et son amplitude, un pointage lui est assigné
    - le type d'effet est reconnaissable par sa couleur :
        - rouge : accroissement de la longueur du serpent
        - orange : réduction de la longueur du serpent

        - magenta : accroissement de la vitesse du serpent
        - violet : réduction de la vitesse du serpent
        - vert : empoisonnement du serpent – ce dernier ne peut plus changer de direction pendant un certain temps
        - bleu : téléportation de la tête du serpent
        - ...
    - prenez note que seules les pastilles rouge et magenta sont obligatoires à réaliser, les autres sont là à titre d'indication
- lorsqu'une entité dynamique se déplace sur la pastille :
    - la pastille est consommée par l'entité dynamique
    - l'entité dynamique est modifiée par l'effet de la pastille
    - l'entité dynamique s'approprie le pointage de la pastille
    - la pastille étant consommée est retirée de la partie
- même si la pastille est généralement présente tant qu'une entité ne l'ait pas consommée, sa durée de vie peut être limitée

Les souris

Les souris sont des entités dynamiques correspondant à un **bonus** pour ce projet. Même si le détail est laissé à votre discrétion, elles doivent représenter ces caractéristiques principales :

- elles sont de la taille d'une seule cellule
- il peut y avoir n_r souris présentes dans l'arène ($n_r \geq 0$)
- elles peuvent être insérées à n'importe quel moment de la partie
- la durée de vie de la souris peut être limitée
- la souris peut consommer des pastilles sans en tirer d'effet ou de pointage
- les serpents peuvent consommer la souris (et profite d'un pointage)
- le déplacement de la souris est déterminé par un algorithme du choix de l'étudiant
- la souris avance inexorablement et ne peut arrêter.

Les serpents

Les serpents sont les entités dynamiques au cœur du projet. Ils sont les seules entités pouvant être contrôlées par le joueur. Ils possèdent ces caractéristiques :

- il peut y avoir entre 1 et 4 serpents simultanés dans l'arène
- les serpents sont constitués d'un corps :
    - le corps est de longueur variable et chaque partie occupe une cellule de l'arène
    - le corps du serpent doit avoir au minimum une longueur de 1
    - il est constitué de 2 parties différentiables par des couleurs différentes :
        - la tête : la première cellule du corps
        - le corps : de la 2^{e} à la dernière cellule
- le serpent possède une orientation courante qui détermine la prochaine position de la tête lors du prochain avancement– seulement les 4 orientations orthogonales sont possibles :
    - vers le haut
    - vers la droite

    - vers le bas
    - vers la gauche
- le serpent avance inexorablement et ne peut s'arrêter
- s'il consomme une pastille ou une souris, il la consume et subit son effet tout en s'attribuant son pointage
- s'il touche à un obstacle, il meurt
- s'il touche à sa propre queue, il meurt
- s'il touche à un autre serpent, il meurt :
    - si les deux serpents ont une collision de tête à tête, les deux serpents meurent
    - sinon, l'autre serpent ne subit aucun dommage
- il possède une vitesse déterminée en cellules par seconde, cette vitesse est déterminée d'avance et peut varier selon l'effet des pastilles
- lorsqu'un serpent change de taille :
    - pour une réduction :
        - on retire instantanément les *n* parties du corps à retirer, de la fin de la queue vers la tête
        - toutefois, la réduction du corps ne peut tuer le serpent et ce dernier ne peut avoir un corps plus petit qu'une seule cellule
    - pour un accroissement :
        - on doit stocker l'information de croissance et faire croître le serpent une cellule à la fois lors des prochains avancements
        - lors d'un *tic*, si le serpent doit croître, on fait avancer sa tête en laissant le bout de sa queue en place
        - ainsi, le serpent s'allonge d'une cellule par *tic* tant qu'il doit croître
- il possède un contrôleur paramétrable permettant de déterminer son orientation
- chaque serpent peut permettre ou restreindre les changements d'orientation de 180 degrés, s'il le permet, le serpent peut mourir instantanément si le joueur donne la mauvaise direction

Chaque serpent doit posséder son propre contrôleur. Minimalement, un contrôleur par clavier est requis, mais il est possible d'avoir une infrastructure plus générique :

- un contrôleur manuel par les touches du clavier :
    - il est possible d'avoir un contrôleur en direction absolue :
        - vers le haut
        - vers la droite
        - vers le bas
        - vers la gauche
    - il est possible d'avoir un contrôleur en direction relative :
        - tourne vers la droite
        - tourne vers la gauche
    - si plusieurs serpents possèdent un tel contrôleur, ils doivent chacun avoir un assignement de touches différent
- un contrôleur par d'autres dispositifs comme la souris de l'ordinateur ou une manette de jeu (en bonus)
- un contrôleur automatique par un algorithme quelconque (en bonus)

Scénarii de partie

L'infrastructure logicielle mise en place permet plusieurs scénarii de jeu différents.

Vous devez implémenter au moins 3 scénarii différents.

Snake Origin

Ce scénario correspond à la version du jeu de base.

- un seul serpent est présent :
  - la tête positionnée au centre de l'arène
  - orienté vers le haut
  - avec une longueur initiale de 3
  - un contrôleur par clavier en direction absolue avec les 4 flèches du clavier
- seul une autre entité peut se retrouver dans l'arène :
  - en tout temps, il doit y avoir une pastille rouge dans l'arène
  - fait croître le serpent de longueur 1
  - donne 1 point
- objectif :
  - on tente de faire le plus de points possible jusqu'à la mort du serpent

Snake Blockade

Ce scénario est similaire à la version du jeu « Blockade »

- deux serpents sont présents :
  - leurs têtes sont positionnées :
    - horizontalement à 1/3 et 2/3 de l'arène
    - verticalement au centre
  - les deux sont orientés vers le haut
  - ils ont une longueur de 3
  - ils possèdent chacun un contrôleur par clavier avec le type de direction au choix :
    - absolue :
      - serpent de gauche utilise les touches W-A-S-D
      - serpent de droite utilise les flèches ou I-J-K-L
    - relatives :
      - serpent de gauche utilise les touches A-D
      - serpent de droite utilise les flèches gauche et droite ou J-L
- une pastille est insérée à toutes les 5 secondes de jeu :
  - le type est déterminé aléatoirement ainsi :
    - 60 % de chance d'avoir une pastille de croissance de longueur aléatoire entre 1 et 10 et de pointage équivalent
    - 40 % de chance d'avoir une pastille d'accélération de vitesse supplémentaire aléatoire entre 2.5% et 5.0%
- objectif :
  - le dernier serpent vivant

Snakify! (scénario personnalisé)

Le troisième scénario est laissé à votre discrétion. On vous demande de créer un scénario original, intéressant et différent des deux premiers.

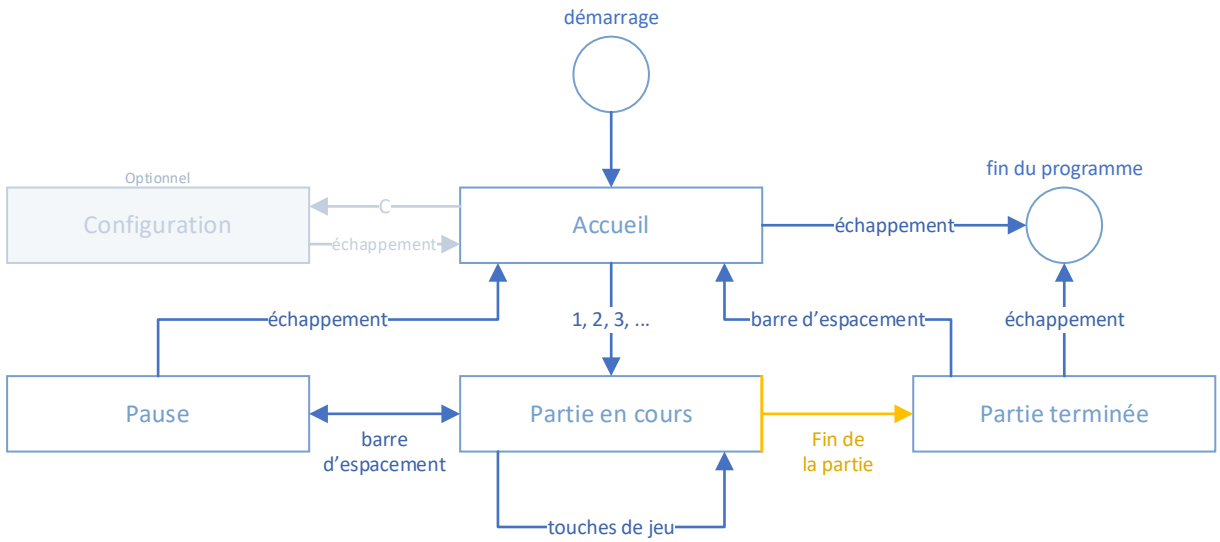
Détails sur le déroulement du logiciel

Le logiciel est constitué de quatre (optionnellement 5) états différents et, à chaque moment, il existe un état courant correspondant à l'un d'entre eux. Pour chacun des états, il existe des évènements permettant d'effectuer des transitions vers d'autres états. Les états sont :

- Accueil
- Partie en cours
- Partie terminée
- Pause
- Configuration (optionnel)

Machine à états finis

Le comportement général du logiciel correspond à une machine à états finis. Le schéma suivant représente cette machine avec ses quatre états et ses onze transitions incluant leur condition.



Il est possible de modifier ces états de façon à améliorer le projet. Toutefois, la mécanique indiquée doit être fonctionnelle d'une façon ou d'une autre.

Implémentation de la machine à états finis

Il existe plusieurs implémentations possibles pour réaliser une machine à états finis.

La première est plus simple, directe, performante, mais aucunement flexible. Elle consiste à définir une énumération représentant chacun des états, à créer une variable du type de cette énumération représentant l'état courant et, à l'aide de switch, gérer explicitement les activités du logiciel en fonction des transitions.

La deuxième est plus sophistiquée, plus abstraite à comprendre, moins facile à mettre en œuvre, mais beaucoup plus flexible et modulaire. Cette deuxième approche peut même, à la limite, être plus rapide à développer en termes de nombre de ligne de code mais beaucoup plus complexe à comprendre. Ainsi, le défi n'est pas la quantité de code à écrire, mais plutôt de comprendre comment le faire intelligemment. En annexe, vous trouverez un diagramme de classes UML présentant les constituants principaux d'une telle implémentation.

Les deux approches sont possibles pour le projet, mais avant d'entreprendre la deuxième, assurez-vous d'en discuter avec les auxiliaires d'enseignement pour comprendre le UML et vous assurer d'avoir une bonne stratégie de développement.

État : Accueil

L'état Accueil est l'état initial et celle active au démarrage du programme. Cet état doit être invitant et permettre une transition facile vers les fonctionnalités importantes du jeu, notamment, le démarrage d'une partie.

Affichage

Le contenu de la page d'accueil est laissé à votre discrétion. Vous avez deux seules contraintes à respecter :

- elle est absente de contenu éthiquement discutable
- vous devez indiquer d'une façon ou d'une autre quelles touches fait quelles actions.

Transitions possibles

Les actions suivantes permettent une transition:

- les touches « 1 », « 2 », « 3 », … : déterminent le type de jeu et débute la partie en passant à l'état *Partie en cours*
- la touche d'« échappement » (« *escape* ») : quitte le programme sans confirmation de l'usager (optionnellement, une confirmation à l'usager est demandée)
- optionnellement, touche « C » : passe à l'état configuration

État : Partie en cours

À l'arrivée dans cet état, une nouvelle partie est démarrée selon le type attendu. Ensuite, le logiciel effectue les *tics* de simulation et les joueurs peuvent jouer.

Affichage

L'affichage est constitué de :

- l'arène et de toutes les entités
- les informations et le pointage

Transitions possibles

Les actions suivantes permettent une transition:

- la touche « barre d'espacement » : passe à l'état *Pause*
- la fin d'une partie : passe à l'état *Partie terminée*

Autres actions

Les serpents sont contrôlés par leur contrôleur. Pour les contrôleurs à clavier, l'appui des touches modifie l'orientation des serpents concernés.

État : Pause

Lorsque le logiciel est sur pause, la partie est suspendue et il n'y a aucun *tic* de simulation.

Affichage

L'affichage est constitué de :

- l'arène et toutes les entités
- les informations et le pointage
- un message mentionnant que la partie est en pause et quelles touches fait quelles actions

Transitions possibles

Les actions suivantes permettent une transition:

- la touche « barre d'espacement » : retourne dans l'état *Partie en cours*
- la touche d'« échappement » : quitte la partie et retourne dans l'état *Accueil*

État : Partie terminée

Lorsqu'une partie est en cours et que les conditions de fin de partie sont respectées, le logiciel bascule automatiquement de l'état *Partie en cours* à l'état *Partie terminée*.

Affichage

L'affichage est constitué de :

- l'arène et ses constituants
- les informations et le pointage
- un message mentionnant :
    - que la partie est terminée
    - indiquant, selon le scénario, le gagnant
    - quelles touches fait quelles actions

Transitions possibles

Les actions suivantes permettent une transition:

- la touche « barre d'espacement » : va dans l'état *Accueil*
- la touche d'« échappement » : quitte le logiciel

État : Configuration (optionnel)

Si vous décidez de réaliser cet état, sachez que son rôle est de configurer les paramètres du jeu. Par exemple, la couleur des serpents, les types de contrôleurs associés ou toute autre option qui est pertinente selon vos scénarii.

Notes sur l'architecture logicielle

L'infrastructure mise en place permet de créer des scénarii de jeux élaborés et variables en permettant des configurations de plusieurs entités.

Toutefois, le projet ne vise pas à réaliser des dizaines de scénarii ou d'entités, mais plutôt de mettre en place une infrastructure relativement flexible assurant une extensibilité remarquable.

L'objectif de cette approche est de valoriser les acquis du cours de GPA434 et de les intégrer avec des considérations algorithmiques et relatives aux structures de données.

Malgré qu'il semble trivial de réaliser un tel jeu, il n'est pas simple de le faire de façon générique et de façon performante.

L'objectif du projet n'est pas simplement de faire un jeu de serpent, mais plutôt de bien le faire en réfléchissant sur tous les aspects de ce dernier. Principalement, une attention particulière est apportée sur l'aspect générique de l'approche, mais aussi, sur la performance des implémentations internes.

Importance d'une approche générique

Le développement d'un projet informatique avec une approche générique présente de nombreux avantages, surtout pour des projets complexes ou qui visent à être extensibles. Ce projet en est un excellent exemple.

En fait, le jeu original du serpent n'a pas besoin d'une telle flexibilité, toutefois, le logiciel que vous devez créer possède beaucoup plus de considérations.

Avantages d'une approche générique

- Flexibilité:
  - Permets de facilement ajouter de nouvelles fonctionnalités et de modifier le comportement du jeu sans avoir à réécrire une grande partie du code.
  - Facilite l'adaptation du jeu à différents types de plateformes et d'appareils.
- Réutilisabilité:
  - Le code générique peut être utilisé pour d'autres projets, ce qui permet de gagner du temps et d'éviter la duplication d'efforts.
  - Facilite la maintenance et la correction de bogues.
- Extensibilité:
  - Permets d'ajouter facilement de nouveaux scénarii de jeu, de nouveaux types de serpents, de nouveaux contrôleurs et de nouveaux types de pastilles.
  - Facilite l'ajout de fonctionnalités multijoueurs et de modes de jeu coopératifs ou compétitifs.
- Meilleure architecture:
  - Encourage une architecture propre et bien documentée, ce qui facilite la collaboration entre les développeurs.
  - Permets de mieux gérer la complexité du projet et d'éviter les problèmes de couplage.

Considérations pour ce projet

Prenons l'exemple de ce projet pour illustrer comment une approche générique peut être bénéfique.

- Scénarios de jeu flexibles:
  - Un système générique permet de facilement implémenter différents scénarii de jeu, comme un mode solo, un mode duo, un mode contre la montre, etc.
  - Le code peut être conçu pour gérer plusieurs serpents simultanément, avec des comportements et des contrôles différents.
- Contrôleurs de serpent flexibles:
  - L'implémentation d'une interface générique pour les contrôleurs de serpent permet d'utiliser différents types de contrôles, comme les touches fléchées, le clavier numérique, la souris, le joystick, ou même l'IA.
  - L'implémentation de contrôleur peut se faire progressivement sans soucis pour l'architecture du code. De nouveaux types de contrôleurs peuvent être ajoutés facilement sans modifier le code principal du jeu.
- Types de pastilles variés:
  - Le système peut être conçu pour gérer différents types de pastilles avec des effets variés, comme des bonus de vitesse, des changements de direction, des points bonus, etc.
  - De nouveaux types de pastilles peuvent être ajoutés facilement pour enrichir les parties.

Difficultés associées à une approche générique

- Complexité accrue:
  - La conception et la mise en œuvre d'un système générique peuvent être plus complexes que celles d'un système spécifique. C'est pourquoi on vous offre une conception déjà faite.
  - Requiers souvent une équipe de développement plus mature pour mieux comprendre les impacts et les coûts d'une telle approche. Il est souvent difficile de trouver un bon équilibre afin d'éviter une trop grande généricité qui est trop coûteuse et inutile.
  - La documentation et la maintenance du code générique peuvent également être plus difficiles.
- Coût initial plus élevé:
  - Le développement initial d'un projet générique peut prendre plus de temps et coûter plus cher que celui d'un projet spécifique.
- Nécessité d'une bonne planification:
  - Une bonne planification et une architecture bien pensée sont essentielles pour garantir le succès d'un projet générique.

En conclusion

L'utilisation d'une approche générique pour le développement d'un projet informatique présente de nombreux avantages, mais elle n'est pas sans difficulté. Il est pratiquement toujours plus payant d'utiliser une telle approche, mais difficile de bien savoir où arrêter. Pour des projets en apparence simples, mais plus complexes, une approche générique peut offrir une grande flexibilité et une extensibilité importante, ce qui permet de créer un jeu riche et évolutif, mais surtout un produit avec plus de valeur.

Algorithmes spécifiques au projet

Défis algorithmiques

Ce projet présente quatre défis algorithmes spécifiques. On vous présente ici chacun d'entre eux appuyé par deux types de solutions, la première, plus facile à réaliser est moins performante alors que la seconde, plus difficile à réaliser, est plus performante.

Gestion du corps du serpent

Le corps du serpent est représenté par une séquence linéaire de segments. La tête du serpent est toujours le premier segment et détermine la position du prochain mouvement en fonction de la direction du serpent. Le défi consiste à gérer le déplacement de chaque segment du corps lorsque le serpent évolue.

Le corps du serpent change selon trois situations:

- Augmentation de la longueur du corps (lorsque le serpent mange la pastille associée)
- Réduction de la longueur du corps (lorsque le serpent mange la pastille associée)
- Déplacement du serpent (à chaque intervalle de temps où il y a déplacement)

Deux approches de complexité différente sont présentées ici pour gérer le corps du serpent, en supposant qu'il y a *n* segments.

Solution O(n)

L'approche la plus simple consiste à utiliser un tableau redimensionnable pour stocker les segments du serpent et effectuer ces opérations:

- Augmentation de la taille du serpent O(n*) :
  - Si la capacité du tableau est insuffisante :
    - Réallouer un nouveau tableau de taille plus grande.
    - Copier tous les segments du corps dans le nouveau tableau.
    - Supprimer l'ancien tableau.
  - Dans tous les cas :
    - Ajouter le nouveau segment à la fin du tableau.
- Réduction de la taille du serpent O(1) :
  - Supprimer le dernier segment du tableau sans modifier sa capacité.
- Déplacement du serpent O(n) :
  - Pour chaque segment s_i du corps, à partir de $i = n - 1$ jusqu'à $i = 1$:

    $$s_i = s_{i-1}$$

  - Définir la position de s_0 (la tête du serpent) à la nouvelle position calculée en fonction de la direction du mouvement.

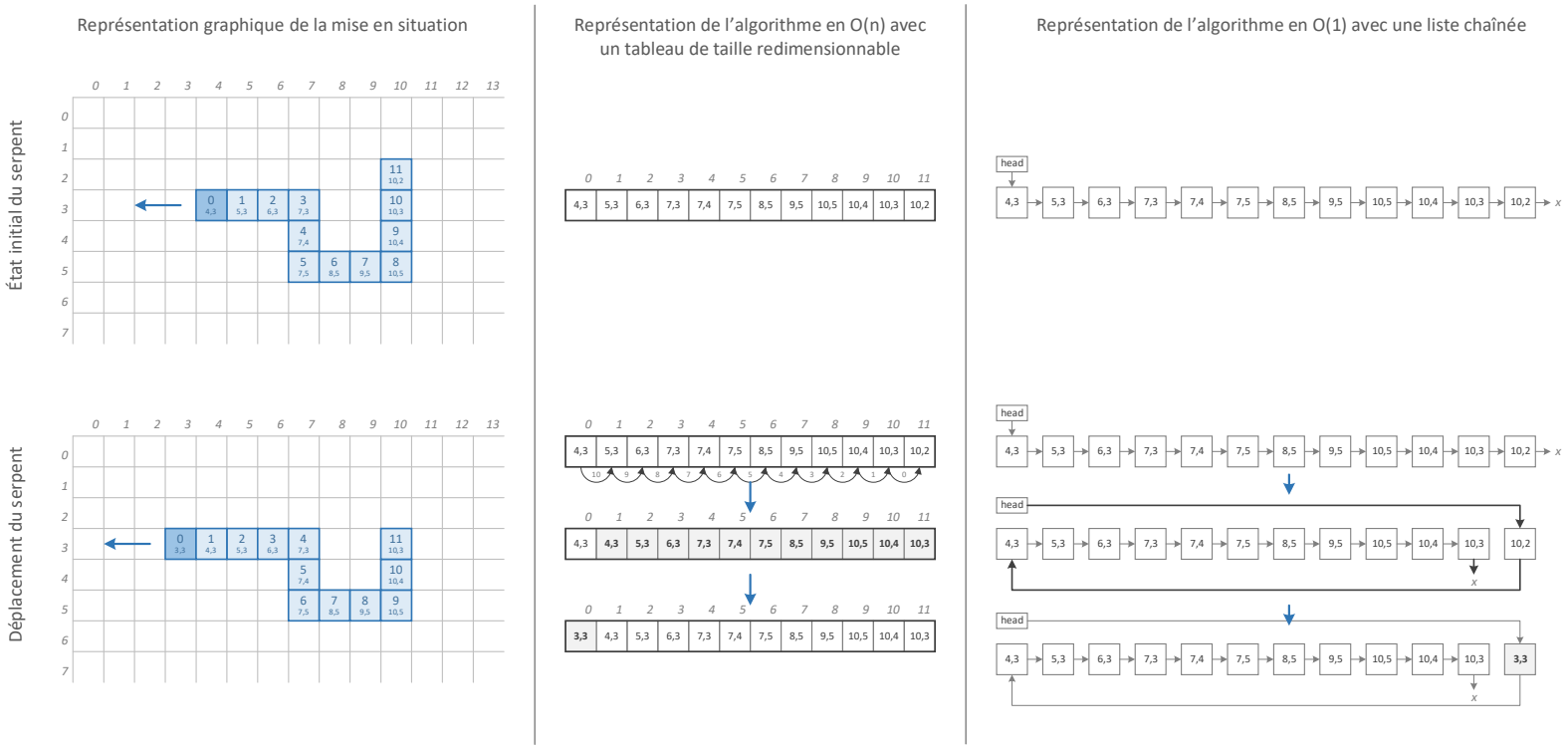
Solution O(1)

L'une des approches possibles réalisant cette tâche en temps constant consiste à utiliser une liste chaînée pour stocker les informations de corps.

- Augmentation de la taille du serpent O(1) :
  - Ajouter les nouveaux segments à la fin de la liste chaînée

- Réduction de la taille du serpent O(1) :
    - Supprimer les segments en trop à la fin de la liste chaînée
- Déplacement du serpent O(1) :
    - Déplacer le nœud du dernier segment (la queue) devant le nœud du premier segment (la tête) dans la liste chaînée
    - Mettre à jour la nouvelle position de la tête en fonction de la direction du mouvement

Représentation graphique



Considérations d'implémentation

Attention, pour le projet, vous ne pouvez pas utiliser l'approche O(n), vous devez implémenter l'approche O(1) puisque le développement de la liste chaînée est au cœur du projet.

Il est important de réaliser que la liste chaînée à réaliser est double alors qu'une version de liste chaînée simple aurait été suffisante. De plus, la liste à réaliser possède plusieurs opérations non essentielles pour le projet. Évidemment, cette implémentation vise l'atteinte de compétences pédagogiques.

Identification d'une position libre pour l'insertion d'entités dans l'arène

Au cours d'une partie, il est parfois nécessaire d'introduire de nouvelles entités dans l'arène, comme une pastille ou d'autres éléments. Cette opération requiert la localisation d'une position libre choisie aléatoirement pour y placer la nouvelle entité. Dans le cadre de ce projet, on se limite à la recherche d'une unique cellule libre, ce qui simplifie considérablement la tâche. Trouver un ensemble de positions libres adjacentes constituerait un défi d'une toute autre ampleur.

Pour traiter cette question, il convient de prendre en compte deux éléments clés :

- Les dimensions de l'arène, soit le nombre total de cellules n_c. Ce nombre est déterminé par le produit des dimensions horizontales (largeur, ou « *width* ») et verticales (hauteur, ou « *height* ») de l'arène. Formellement, on a : $n_c = width \times height$.
- Le nombre de cellules occupées par un segment d'entité, désigné par n_s. Pour chaque entité, on compte les segments qui en occupent l'espace. Par exemple, une pastille utilise une cellule alors qu'un serpent utilise autant de cellules qu'il a de segments.

Solution O(n)... pas tout à fait

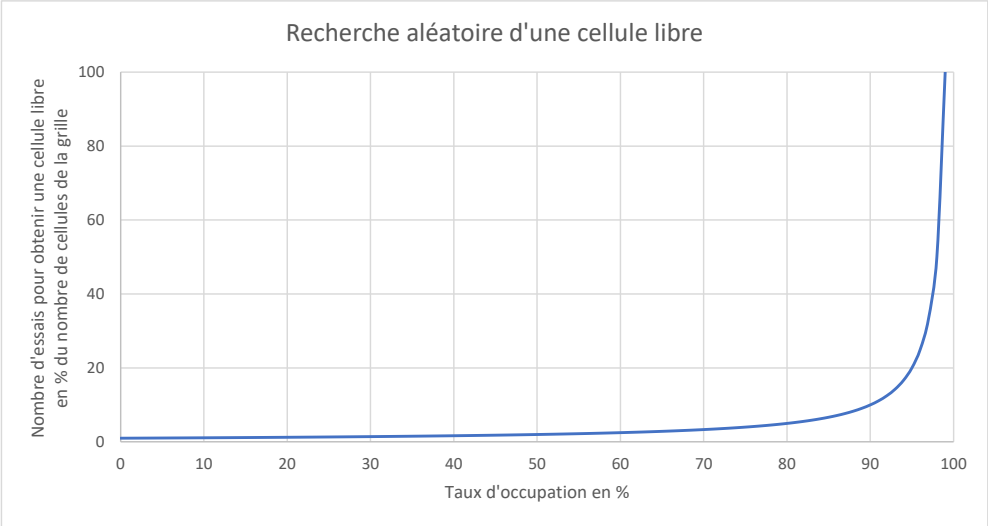
Pour identifier aléatoirement une cellule libre dans une grille, on peut utiliser un algorithme basé sur un générateur de nombres aléatoires suivant une distribution uniforme. Chaque cellule, par conséquent, a une probabilité égale d'être choisie. L'algorithme procède comme suit :

1. Générer deux nombres aléatoires uniformes, x et y correspondant aux dimensions horizontale et verticale de la grille.
2. Vérifier si la cellule de coordonnées (x, y) est libre.
3. Si la cellule est occupée, recommencer à l'étape 1 jusqu'à ce qu'une cellule libre soit trouvée.
4. Si la cellule est libre, l'algorithme s'arrête et retourne les coordonnées de la cellule.

Cette méthode présente cependant deux problématiques principales :

- Premièrement, quelle est la complexité de la vérification de disponibilité d'une cellule à l'étape 2. Si cette vérification est réalisée au moyen d'une liste contenant toutes les entités, le temps nécessaire pour confirmer la disponibilité d'une cellule est de l'ordre de $O(n_s)$.
- Deuxièmement, le nombre d'itérations nécessaires pour trouver une cellule libre peut être important. Ce nombre est proportionnel au rapport entre les cellules occupées n_s et le nombre total de cellules n_c, soit le ratio d'occupation $\frac{n_s}{n_c}$. Plus le nombre de cellules libres est grand, moins on aura besoin d'essais pour en localiser une.

Dans un cadre où chaque tentative est un essai de Bernoulli indépendant avec une probabilité de succès p et d'échec de $1 - p$, l'espérance mathématique du nombre d'essais nécessaires pour trouver une cellule libre est $\frac{1}{p}$. Dans notre cas, p est défini par $p = \frac{n_c - n_s}{n_c}$. Le graphique suivant illustre le nombre d'essais requis en fonction du taux d'occupation des cellules.



En conclusion, l'ordre de complexité est formellement $O\left(\frac{n_c}{n_c - n_s}\right)$. Dans le scénario le moins favorable, où la grille ne possède qu'une seule cellule libre, l'algorithme peut atteindre une complexité de $O(n_c)$, ce qui est le nombre total de cellules dans la grille.

Solution O(1)

Il existe plusieurs approches pour résoudre ce problème.

Certaines approches sont basées sur des structures de données permettant des temps d'accès en O(1). Pour ce projet vous **ne pouvez pas** les utiliser car nous les verrons à la fin de la session. Il s'agit de `std::unordered_set` et `std::unordered_map`.

Il existe une autre approche permettant de résoudre le problème en temps constant en maintenant à jour une séquence linéaire indiquant où sont les cellules libres. Avec cette approche, il est important de garder en tête que pour obtenir un temps final en temps constant, plusieurs opérations doivent être en O(1) :

1. identification d'une cellule libre
2. retirer de la séquence des cellules libre une cellule qui vient d'être déclarée occupée
3. ajouter dans la séquence des cellules libre une cellule qui vient d'être déclarée libérée

Le défi est intéressant et se réalise par une cascade de deux tableaux contigus en mémoire.

Le premier tableau, nommé `emptyCells`, contient la liste de toutes les cellules. Chaque cellule contient la position d'une cellule du tableau original. Un index supplémentaire permet de déterminer un point pivot, nommé `p`, indiquant une séparation du tableau où, `< p`, se trouvent les cellules libres et, `≥ p`, se trouvent les cellules occupées.

Ce tableau permet les étapes 1 et 2 en temps constant :

1. Pour l'étape 1, il suffit de générer un nombre aléatoire entre 0 et `p-1`, nommé `v`. La cellule à la position `v` représente une cellule disponible pour l'insertion.

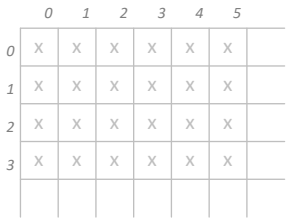
2. Pour l'étape 2, il suffit de permuter les valeurs des positions indiquées par les index **v** et **p-1**, puis décrémenter **p**.

Le problème avec cette approche est que la tâche 3 est impossible en temps constant. Pour y remédier, il convient de créer un deuxième tableau, nommé **cellIndices**, indiquant où se trouvent chacune des cellules originales dans **emptyCells**. Les tâches à réaliser sont :

2. Avec l'usage de ce tableau supplémentaire, il requiert de modifier l'étape 2 afin ajouter une maintenance du tableau **cellIndices** en fonction des opérations sur **emptyCells**.
3. Dans **emptyCells**, permutation des valeurs aux positions où trouve la libération et où se trouve la cellule rendu disponible dans **emptyCells**. Cette dernière position est retrouvée en O(1) grâce à l'information disponible dans **cellsIndices** (nécessite une double indexation). Mettre à jour **cellsIndices** et incrémenter **p**.

La difficulté de cette approche réside dans le détail d'implémentation et d'un débogage soigné. Néanmoins, on remarque que pour obtenir une complexité en temps $O(1)$ il y a un coût de complexité d'espace de $O(2n_c)$ seulement pour cette tâche.

Représentation graphique

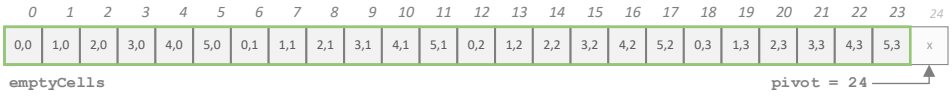


État initial

Initialisation des tableaux emptyCells et cellsIndices

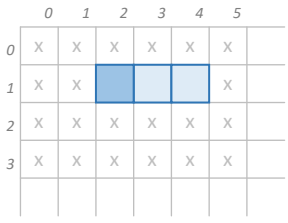
À gauche, représentation de l'état du jeu :
- Les x indiquent les cellules libres
- Les cellules colorées indiquent les cellules occupées

En bas, les tableaux de gestion :
- En gris, les cellules qui ont été modifiées
- En vert, la zone des cellules libres
- En magenta, la zone des cellules occupées



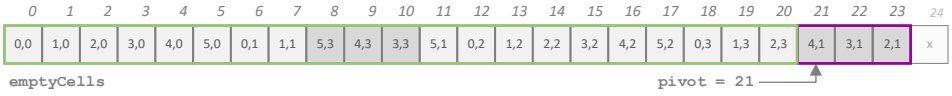
emptyCells — pivot = 24

cellsIndices



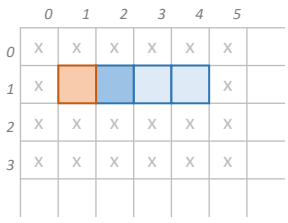
Insertion d'un serpent de taille 3

Mise à jour de emptyCells (permutation) et de cellsIndices



emptyCells — pivot = 21

cellsIndices

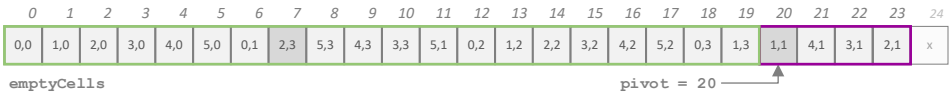


Insertion aléatoire d'une pastille

Génération aléatoire d'une position

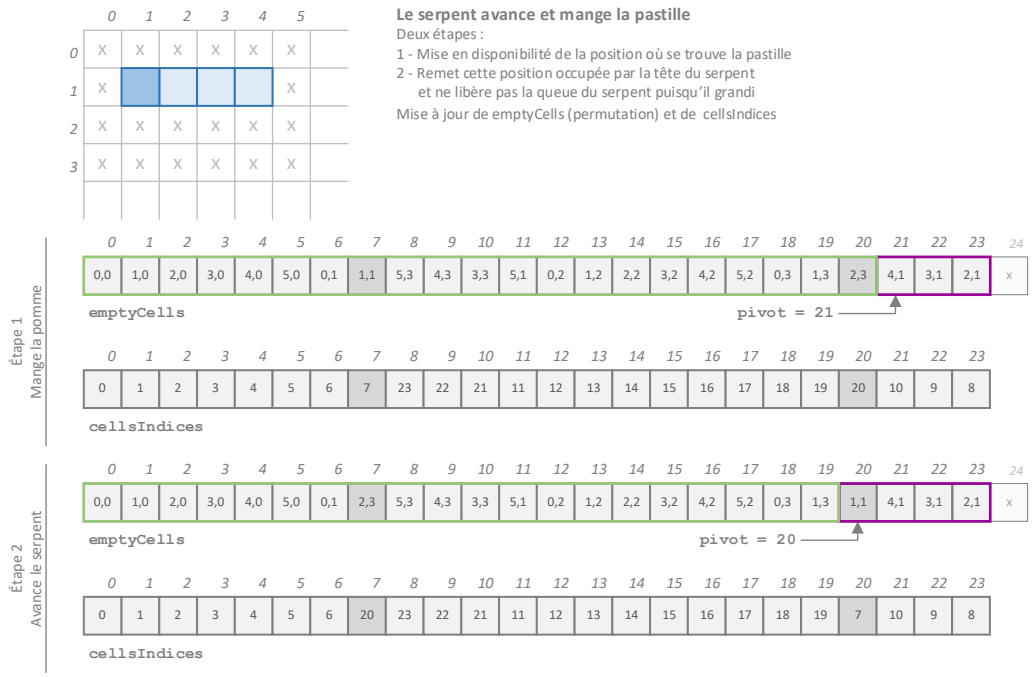
Mise à jour de emptyCells (permutation) et de cellsIndices

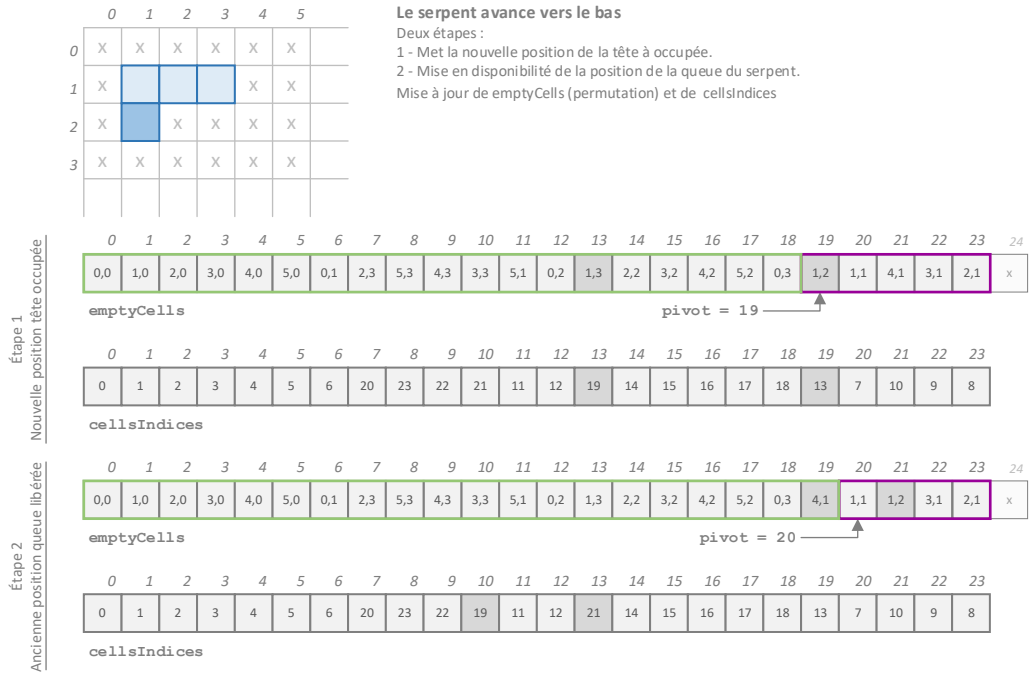
Pour cet exemple, on a générer une valeur aléatoire entre 0 et 20. Par hasard, l'index 7 a été généré.



emptyCells — pivot = 20

cellsIndices





Détection de collision

Lorsqu'un serpent se déplace, il faut déterminer si une collision a lieu. La collision entre deux entités peut avoir lieu entre un serpent et son corps, un autre serpent ou une pastille.

Pour cet algorithme, il faut considérer le nombre de segments n_s utilisés par toutes les entités.

Solution O(n)

L'approche la plus simple est de parcourir la liste d'entités et valider si la nouvelle position du segment de la tête du serpent est en collision avec un autre segment d'une autre entité.

Cette approche, très simple à implémenter, est de complexité $O(n_s)$.

Solution O(1)

Une façon de résoudre ce défi algorithmique en temps constant consiste à créer une grille représentant chaque cellule de l'arène où, chaque possède un pointeur :

- si la cellule est occupée par un segment, elle pointe vers l'entité concernée
- si la cellule est libre, elle est mise à **nullptr**

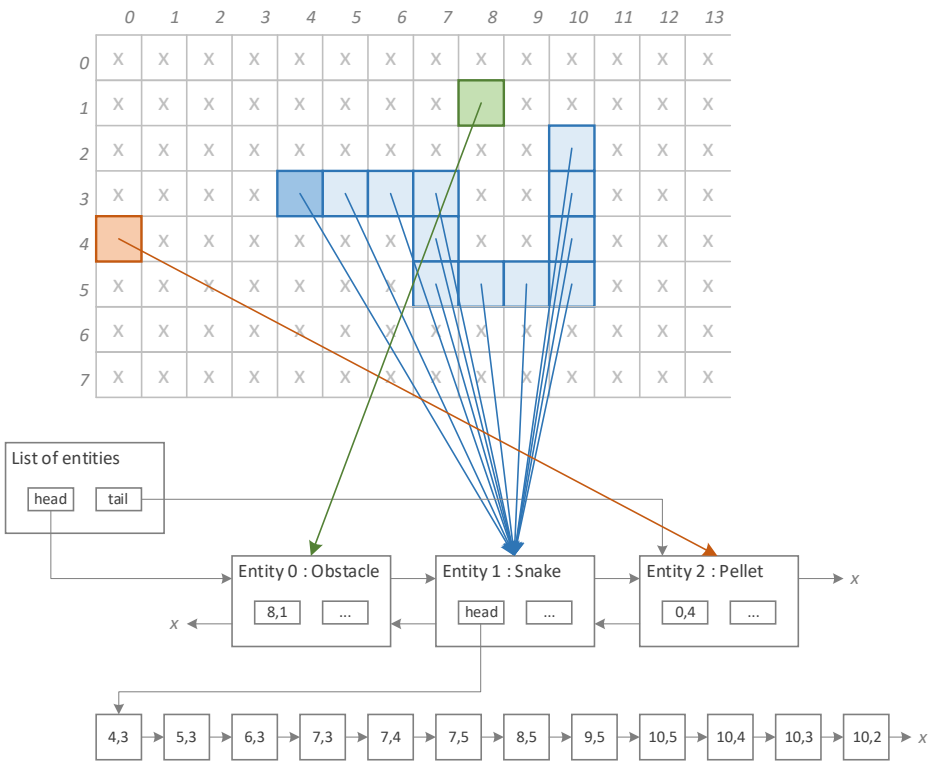
Le défi algorithmique consiste à maintenir à jour en tout temps cette grille en créant des fonctions d'inscription et de désinscription des entités venant occuper ou libérer les cellules. Attention, ce développement est plus difficile qu'il n'y parait. Ces fonctions doivent être réalisées soigneusement et être utilisées avec autant d'attention par les entités au moment opportun. Ce n'est pas difficile en soi, mais c'est un défi qui prend un certain temps pour gérer les cas limites et résoudre les problématiques de synchronicité.

Avec cette approche, il est très simple et rapide de valider s'il y a une collision et, le cas échéant, avec la collision a lieu.

Finalement, on remarque que la réalisation en $O(1)$ pour la détection de collision possède plusieurs similarités avec l'algorithme de recherche d'une position libre aléatoire. Principalement, la mise à jour de structures externes pour chaque opération. Si vous décidez d'effectuer ces deux implémentations, il est possible de les réunir logiquement et intelligemment pour simplifier votre structure logicielle.

Représentation graphique

Représentation synchrone de la liste d'entités et de la grille de cellules occupées et libres



Détection des touches au clavier

Il existe plusieurs mécanismes permettant de détecter les touches du clavier et d'en faire une action. Toutefois, considérant que n_t touches peuvent être appuyés à un certain moment de la partie, l'approche donnée en exemple est simple, mais produit une complexité $O(n_t)$. Il est évident que ce problème n'en est pas un significatif étant donné que la boucle d'exécution est tellement rapide que $n_t$ne sera jamais très grand. En fait, il sera très petit. Néanmoins, il est possible, par diverses approches, de réduire la complexité à $O(1)$.

Pour cette partie du projet, on ne vous donne pas plus de détails sur l'approche à utiliser. C'est à vous d'en proposer une et de la mettre en place.

Mise en garde sur les aspects multimédias

Le développement d'un jeu implique généralement une attention significative aux composantes multimédias, notamment les éléments graphiques et sonores. De plus, malgré le fait que Qt ne soit pas une bibliothèque de développement de jeu vidéo, il est possible d'élaborer des visuels sophistiqués et d'intégrer l'audio de manière relativement aisée.

Cependant, il est important de noter que ces aspects ne constituent pas le cœur du projet. L'objectif n'est pas à ce niveau, c'est davantage l'architecture logicielle et, surtout, les performances liées aux algorithmes qui sont au premier plan. On vous demande de simplement créer des graphismes nets et structurés, sans viser une esthétique de niveau supérieur. Sur le plan sonore, aucune exigence n'est formulée.

Le projet requiert déjà un investissement considérable en termes de développement, et consacrer du temps à améliorer le « *gameplay* » par le cosmétique serait inopportun. Il est à souligner que les auxiliaires d'enseignement ne fourniront aucun soutien pour ces éléments, et qu'aucun point boni ne sera attribué pour l'embellissement des aspects graphiques ou sonores.

Projet de démarrage

On met à votre disposition un projet de démarrage visant à faire une simple démonstration visant :

- à démarrer un projet Qt
- à établir une boucle de contrôle
- à calculer le temps qui s'écoule
- à capturer les informations du clavier (par polymorphisme et une structure de données d'accumulation des touches)
- à dessiner à l'écran

Attention, votre projet est beaucoup plus élaboré et complexe que ce simple exemple. Ce dernier vise uniquement à vous donner un exemple utilisant les techniques de base nécessaires à la mise en place d'un tel logiciel. De plus, vous ne pouvez pas utiliser ce logiciel comme base formelle. Vous devez créer votre propre projet.

Il est important d'assister à la présentation faite par les auxiliaires d'enseignement pour bien comprendre cet exemple.

Développement à réaliser

Vous devez développer un jeu correspondant au concepts présentés tout en suivant (en partie) la conception UML donnée en annexe dans ce document.

Malgré le fait que vous avez beaucoup de liberté, vous devez tout de même respecter les contraintes suivantes :

- Vous devez réaliser votre travail en équipe de 3 ou 4
- Vous devez programmer en C++ avec la librairie Qt
- Vous devez mettre l'emphase sur :
    - la performance des algorithmes
    - un bon niveau de qualité logiciel basé sur le paradigme orienté objet
- La conception UML donnée est partiellement imposée. Vous trouverez en annexe tous les détails à cet effet.
- Vous devez **obligatoirement** créer votre propre classe de gestion du corps du serpent qui est basé sur une liste chaînée double:

    - cette classe doit être réalisée entièrement par vous sans dépendance externe (autre que les éléments **QPoint** nécessaires au projet)
    - vous devez offrir *minimalement* **toutes** les fonctionnalités présentées dans le diagramme de classe de l'annexe 2 (voir la classe **Body** de **Snake**) même si vous ne les utilisez pas dans le reste du projet
    - l'infrastructure fondamentale de cette classe est une liste chaînée double
- Vous devez **obligatoirement** utiliser la classe **std::list** pour gérer toutes les entités de la simulation.
- Sans être obligatoire, vous pouvez utiliser les classes **std::array**, **std::vector**, **std::deque**, **std::queue**, **std::stack**, **std::forward_list** et **std::list** pour toute autre partie du projet. Le choix est laissé à votre discrétion. Toutefois, vous devez le justifier clairement en mettant des commentaires là où vous déclarez la variable associée.
- Il est interdit d'utiliser n'importe quelle autre structure de données (par exemple **std::set**, **std::map**, **QList**, **QSet**, **QMap** ou de nombreux autres).
- Vous devez mettre en pratique autant que possible de bonnes techniques de programmation (**CALTAL**, **DRY**, **OFOT** et **UMUD**).

Rapport

Vous devez créer un document intitulé **rapport.txt** répondant à ces questions. Votre texte doit être très technique, concis et précis.

- Indiquer les noms des étudiants formant l'équipe
- Y a-t-il des fonctionnalités qui n'ont pas été entièrement réalisée.
- Donnez une description du scénario personnalisé.
- Pour chacun des algorithmes mentionnés, discuter de votre implémentation en expliquant quelle approche vous avez utilisé en précisant les adaptations que vous avez faites.
    1. Gestion du corps du serpent.
    2. Identification d'une position libre pour l'insertion d'entités
    3. Détection de collision
    4. Détection des touches au clavier
- Concernant votre architecture logicielle, donnez les points saillants de votre implémentation. Donner une description ultra compacte sous forme de puces.
- Y a-t-il d'autres éléments de votre implémentation sur lesquels vous aimeriez porter une attention.

Annexe: Diagrammes de classes UML

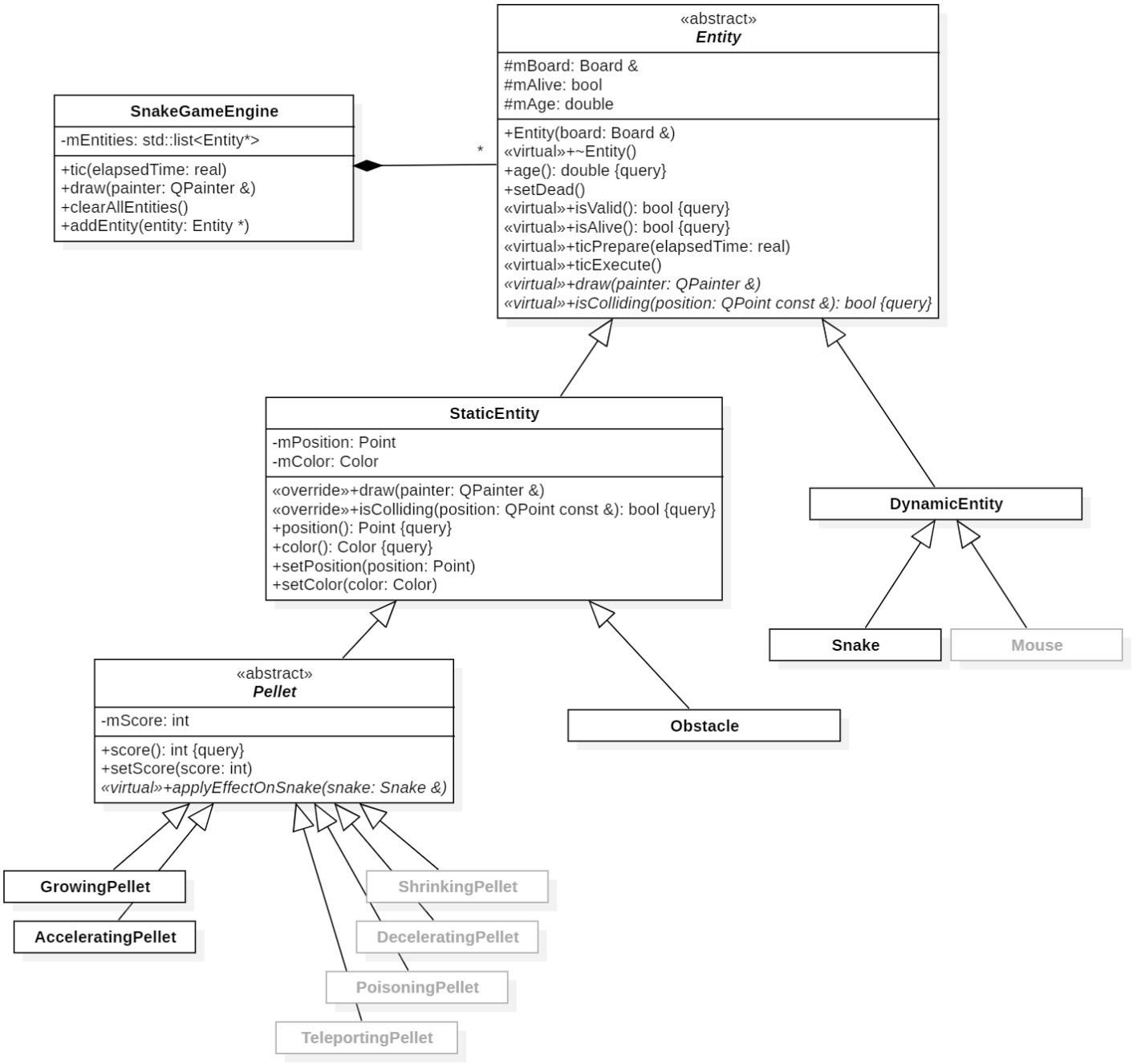
Les diagrammes UML donnés sont présents à titre de référence pour vous inspirer et constituent un exemple de bonne pratique. Il existe d'innombrables autres approches pertinentes, mais l'approche présentée exploite les avantages du paradigme orienté objet tel qu'il a été mis en pratique dans le cours de GPA434.

Sachez que la programmation d'un jeu vidéo repose généralement sur un amalgame de paradigmes de programmation et que l'orienté objet n'en est qu'un parmi d'autres. Néanmoins, dans le cadre de ce cours, on tente de trouver un équilibre entre performance et une qualité logicielle soutenue par le développement orienté objet, valorisant ainsi vos acquis antérieurs en vous donnant l'opportunité de les mettre en pratique dans le contexte d'un projet à réaliser entièrement.

Vous remarquerez que certains diagrammes UML sont à réaliser telle quelle, d'autres sont fortement encouragés alors que les derniers ne sont que des suggestions. Vous trouverez sous les diagrammes les indications de réalisation.

Sauf pour la classe **Snake** et ses dépendances, toutes les classes sont sommairement décrites. Il manque beaucoup d'attributs et d'opérations telles que des accesseurs et des mutateurs. L'objectifs des diagrammes sont de vous donner une ligne directrice des éléments importants vous permettant d'amorcer une infrastructure intéressante. Il vous appartient de les adapter à vos implémentations personnelles.

Entités



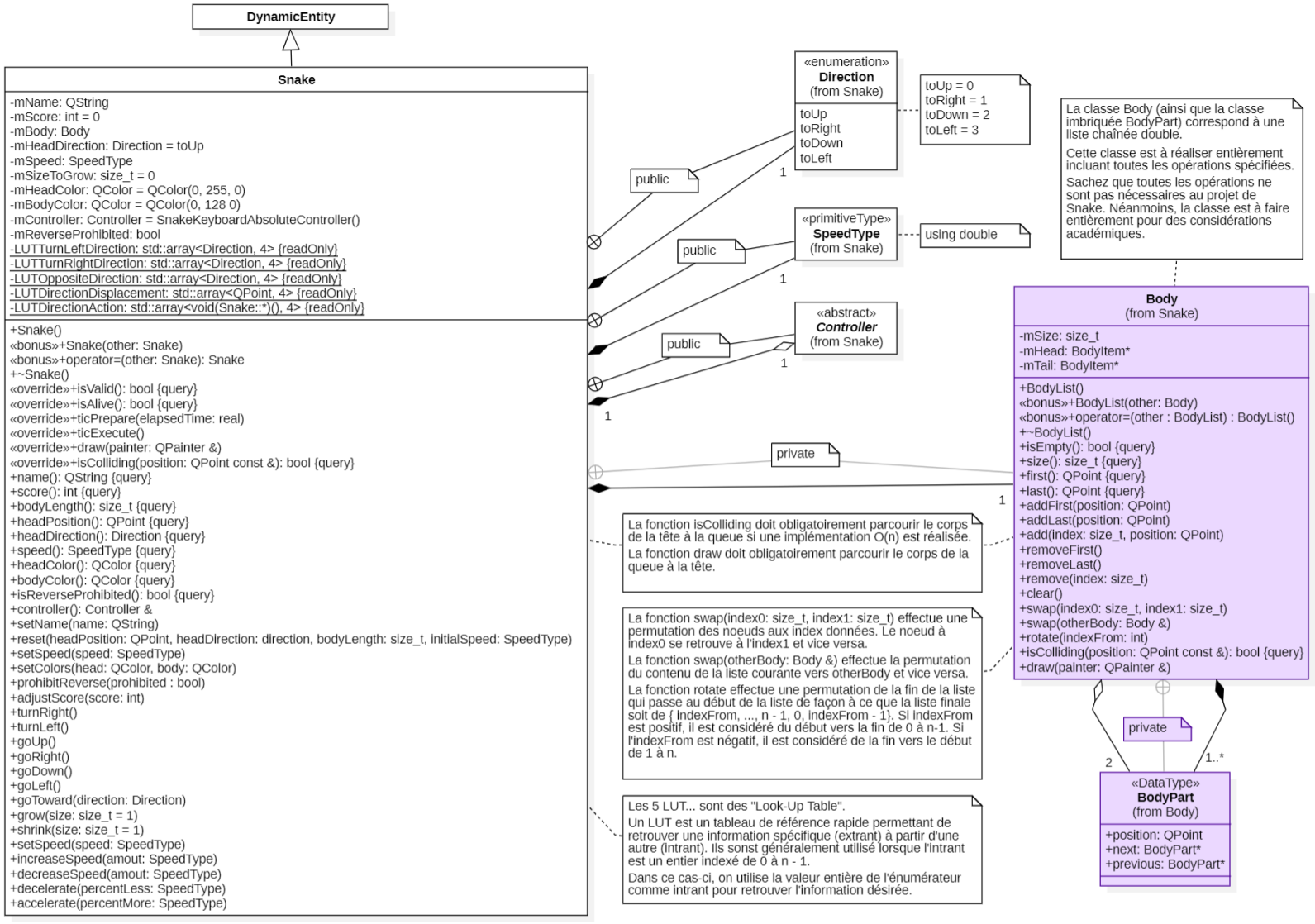
Structure générale des entités

Conception obligatoire, mais adaptable

Plusieurs éléments sont manquants

On présente la plupart des parties essentielles afin de mieux comprendre les rôles et certains détails.

Détails sur pour le serpent

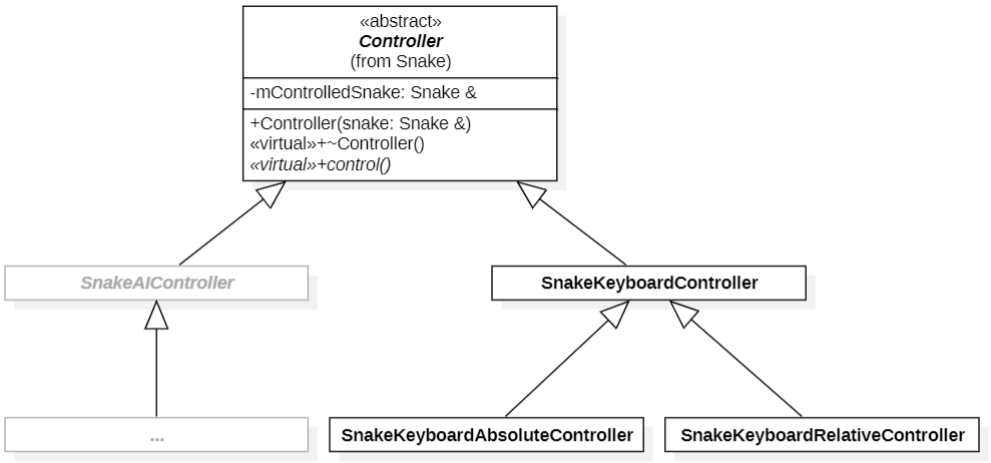


Conception de la classe Snake et de tous ses sous-constituants

Conception obligatoire adaptable

Body et BodyPart sont non adaptables et doivent être réalisée entièrement tel que définis

Détails pour le contrôleur de serpent

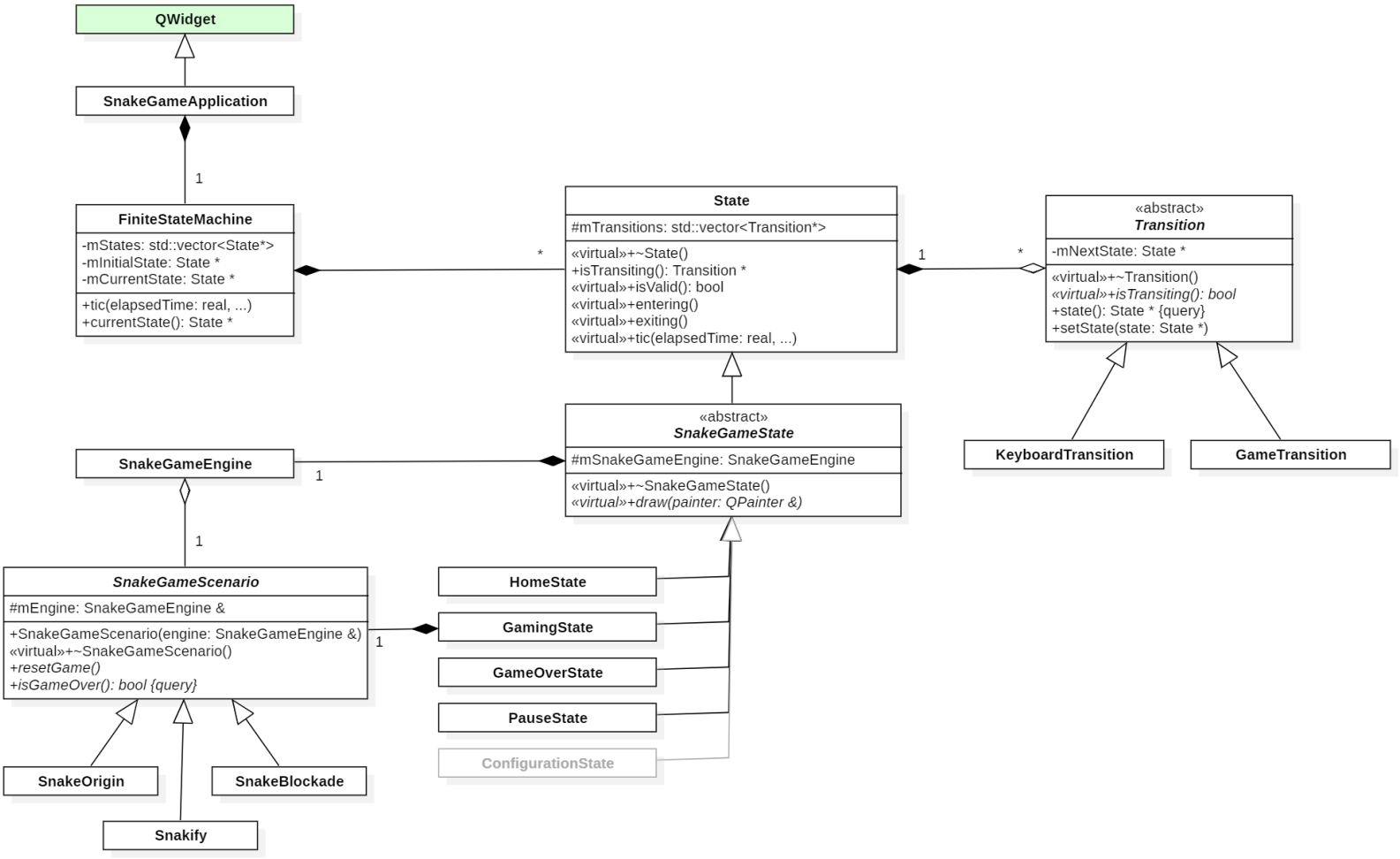


Conception générale des contrôleurs du serpent

Conception recommandée

La conception est vague et manque plusieurs éléments qui doivent être complétés

Structure du logiciel basée sur une machine à états finis



Conception générale de l'architecture
Incluent les grandes lignes d'une machine à états finis
Conception suggérée (plus abstraite à réaliser, mais pas obligatoire)
La conception est vague et manque plusieurs éléments qui doivent être complétés