Développement d'un Jeu Vidéo : City Builder

Programmation Avancée en style Fonctionnel

26 mai 2024

Malek Bouzarkouna 28706508
Paul Mekhail 28708078

Introduction

Ce projet est un devoir de fin d'année réalisé dans le cadre de l'unité d'enseignement "Programmation Avancée en style Fonctionnel" (PAF) du master d'informatique de Sorbonne Université. L'objectif est de réaliser un jeu vidéo de type *city builder* sur une grille, similaire à Sim City (Maxis, 1989) dans le langage Haskell. Afin de proposer une expérience visuelle fluide et intuitive au joueur, la bibliothèque graphique SDL2 est utilisée.

Concrètement, le jeu se déroule sur une carte vue du dessus, où le haut de l'écran représente le Nord. Le joueur peut placer sur la carte des infrastructures (routes, bâtiments, ...) afin de faire évoluer leur ville. Une simulation gère en temps réel la vie des citoyens de la ville individuellement.

Pour minimiser les erreurs logicielles, des propositions d'invariants, pré-conditions et post-conditions permettent de contrôler les entrées / sorties, et le comportement des fonctions utilisées en réalisant des tests générés par *QuickCheck*. De plus, des tests unitaires pertinents sont réalisés avec *HSpec*.

Les règles ont été particulièrement inspirées de ressources extérieures : les critères de construction étaient bien détaillés dans [1], et les explications sur le cycle des habitants étaient particulièrement bien expliqués dans [2]. De plus, ChatGPT a été utilisé afin de réaliser le panel de construction, quelques fonctions complexes (comme la connexité des routes), et la base de test.

1 Manuel d'Utilisation

Le jeu se lance depuis le projet *stack* en tapant la commande `stack run`. La simulation débute. Le joueur dispose alors devant lui d'une grille 1920x1080 sur laquelle il peut faire évoluer sa ville (Figure 1). Le menu en haut à droite de l'écran permet au joueur de bâtir des infrastructures, mais aussi d'obtenir des informations complémentaires comme l'argent à sa disposition, le nombre de résidents, la pollution et la sécurité de sa ville. Lorsque le joueur réalise une action, une fenêtre textuelle apparaît sur la partie supérieure de l'écran pour décrire si l'action a bien été réalisée, ou les raisons pour lesquelles elle a échoué.

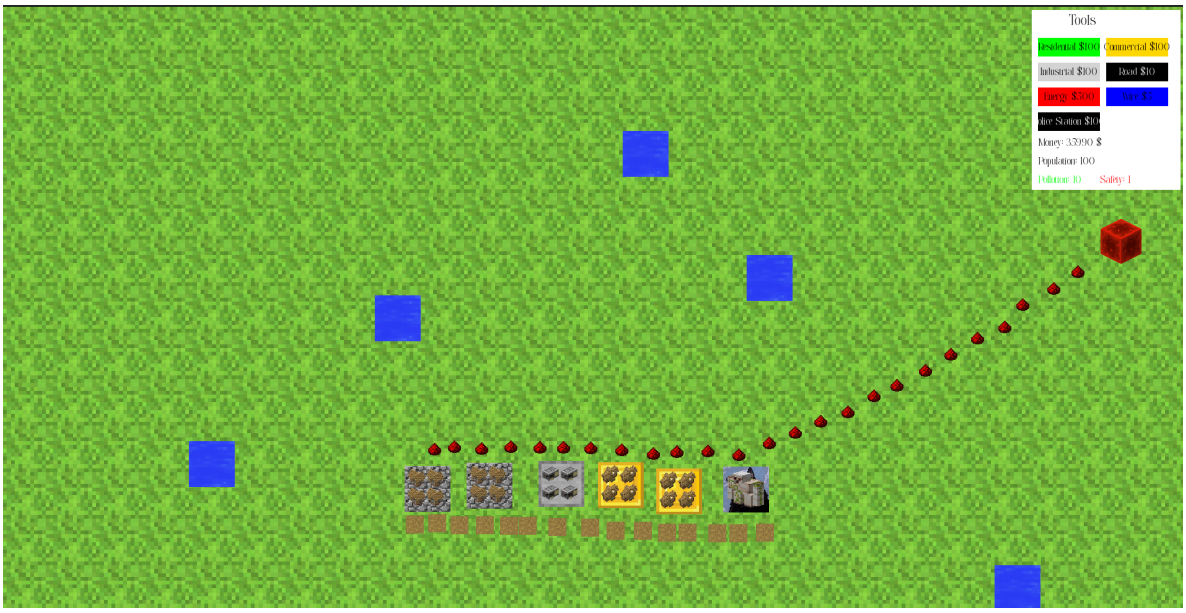

FIGURE 1 – Exemple de simulation

2 Liste des Propositions

Afin d'assurer que les paramètres des différentes fonctions, ainsi que leur comportement, soient cohérents et corrects, nous avons proposé une liste de propositions de propriétés pré-conditions / post-conditions / invariant.

2.1 Propositions associées aux Types

`Ville.hs | Ville`

Invariant (`villeInvariant :: Ville -> Bool`)
Invariant qui vérifie l'absence de collisions, la connectivité des routes pour les zones résidentielle / commerciale / industrielle / administrative, que les zones sont disjointes deux-à-deux, et les fils ou zones d'énergie connectés.

`Ville.hs | Citoyen`

Pré-conditions (`preCitoyen :: Citoyen -> Bool`)
Vérifie que les coordonnées sont non négatives et que les valeurs de l'état sont comprises entre 0 et 100 lors de la création.
Post-conditions (`postCitoyen :: Citoyen -> Citoyen -> Bool`)
Vérifie que les coordonnées sont non négatives et que les valeurs de l'état sont comprises entre 0 et 100 après la création.

`Ville.hs | Occupation`

Pré-conditions (`preOccupation :: Occupation -> Bool`)
Aucune précondition spécifique, les valeurs doivent appartenir à l'enum `Occupation`.
Post-conditions (`postOccupation :: Occupation -> Occupation -> Bool`)
Aucune post-condition spécifique, les valeurs doivent appartenir à l'enum `Occupation`.
Invariant (`invariantOccupation :: Occupation -> Bool`)
Vérifie que les occupations sont valides (pas de vérification supplémentaire requise pour l'enum).

`Shapes.hs | Coord`

Pré-conditions (`preCoord :: Int -> Int -> Bool`)
Vérifie que les coordonnées sont non négatives lors de la création.
Post-conditions (`postCoord :: Int -> Int -> Coord -> Bool`)
Vérifie que les coordonnées créées sont non négatives.
Invariant (`invariantCoord :: Coord -> Bool`)
Vérifie que les coordonnées sont non négatives.

`Shapes.hs | Forme`

Pré-conditions (`preForme :: Forme -> Bool`)
Vérifie que les dimensions sont positives lors de la création.
Post-conditions (`postForme :: Forme -> Forme -> Bool`)
Vérifie que les dimensions de la forme créée sont positives.
Invariant (`invariantForme :: Forme -> Bool`)
Vérifie que les dimensions des formes sont positives.

Shapes.hs | Zone

Pré-conditions (`preZone :: Zone -> Bool`)
Vérifie que la forme de la zone est valide et que les bâtiments (s'il y en a) ont des formes valides lors de la création.
Post-conditions (`postZone :: Zone -> Zone -> Bool`)
Vérifie que la zone créée est valide.
Invariant (`invariantZone :: Zone -> Bool`)
Vérifie que la forme d'une zone est valide et que les bâtiments (s'il y en a) ont des formes valides.

Shapes.hs | Batiment

Pré-conditions (`preBatiment :: Batiment -> Bool`)
Vérifie que la forme d'un bâtiment est valide et que les coordonnées sont non négatives lors de la création.
Post-conditions (`postBatiment :: Batiment -> Batiment -> Bool`)
Vérifie que le bâtiment créé est valide.
Invariant (`invariantBatiment :: Batiment -> Bool`)
Vérifie que la forme d'un bâtiment est valide et que les coordonnées sont non négatives.

2.2 Propositions associées aux Opérations

Ville.hs | addBatiment :: Batiment -> Ville -> ZonId -> Ville

Pré-conditions (`preAddBatiment :: Batiment -> Ville -> ZonId -> Bool`)
Le bâtiment doit être ajouté à une zone résidentielle, industrielle ou commerciale.
Post-conditions (`postAddBatiment :: Batiment -> Ville -> ZonId -> Ville -> Bool`)
Le bâtiment est ajouté à la zone spécifiée dans la ville.
Description
Ajoute un bâtiment à une zone.

Ville.hs | isPowered :: Ville -> Zone -> Bool

Pré-conditions (`preIsPowered :: Ville -> Zone -> Bool`)
Aucune.
Post-conditions (`postIsPowered :: Ville -> Zone -> Bool -> Bool`)
Retourne True si la zone est alimentée, False sinon.
Description
Vérifie si une zone est alimentée.

Ville.hs | isWire :: Zone -> Bool

Pré-conditions (`preIsWire :: Zone -> Bool`)
Aucune.
Post-conditions (`postConstruit :: Ville -> Zone -> ZonId -> Ville -> Bool`)
Retourne `True` si la zone est un fil, et `False` sinon.
Description
Vérifie si une zone est un fil ou non.

`Ville.hs | construit :: Ville -> Zone -> ZonId -> Ville`

Pré-conditions (`preConstruit :: Ville -> Zone -> ZonId -> Bool`)
Aucune.
Post-conditions (`postConstruit :: Ville -> Zone -> ZonId -> Ville -> Bool`)
La zone doit être ajoutée à la ville avec l'ID de zone spécifié.
Description
Ajoute une zone à la ville.

`Ville.hs | assignImmigrantToZones :: Ville -> Citoyen -> Maybe Citoyen`

Pré-conditions (`preAssignImmigrantToZones :: Ville -> Citoyen -> Bool`)
Aucune.
Post-conditions (`postAssignImmigrantToZones :: Ville -> Citoyen -> Maybe Citoyen -> Bool`)
Si des zones résidentielles, industrielles et commerciales sont disponibles, l'immigrant devient un habitant avec des assignments à ces zones. Sinon, l'immigrant reste inchangé.
Description
Transforme un immigrant en citoyen si les critères requis sont satisfaits, en lui attribuant une zone résidentielle, une zone industrielle et une zone commerciale.

`Ville.hs | updateCitizens :: Ville -> Ville`

Pré-conditions (`preUpdateCitizens :: Ville -> Bool`)
Aucune.
Post-conditions (`postUpdateCitizens :: Ville -> Ville -> Bool`)
Met à jour les positions et occupations des citoyens dans la ville.
Description
Actualise tous les attributs des citoyens, notamment leurs coordonnées et l'activité qu'ils sont en train de faire.

`Ville.hs | setOccupation :: Citoyen -> Occupation -> Citoyen`

Pré-conditions (`preSetOccupation :: Citoyen -> Occupation -> Bool`)
Aucune.
Post-conditions (`postSetOccupation :: Citoyen -> Occupation -> Citoyen -> Bool`)
La nouvelle occupation est assignée au citoyen.
Description
Modifie l'activité en cours du citoyen.

`Ville.hs | mettreAJourPosition :: Ville -> Citoyen -> Citoyen`

Pré-conditions (`preMettreAJourPosition :: Ville -> Citoyen -> Bool`)
Aucune.
Post-conditions (`postMettreAJourPosition :: Ville -> Citoyen -> Citoyen -> Bool`)
La position et l'occupation du citoyen sont mises à jour en fonction de l'état actuel de la ville.
Description
Met à jour les coordonnées (la position) d'un citoyen sur la grille.

`Ville.hs | aStar :: Ville -> ZonId -> ZonId -> Maybe Path`

Pré-conditions (`preAStar :: Ville -> ZonId -> ZonId -> Bool`)
Les IDs de zones de départ et d'arrivée doivent être valides.
Post-conditions (`postAStar :: Ville -> ZonId -> ZonId -> Maybe Path -> Bool`)
Retourne le chemin optimal entre les deux zones s'il existe, sinon retourne `Nothing`.
Description
Implémente l'algorithme de parcours de graphe A^*

`Ville.hs | calculateRent :: Batiment -> Int`

Pré-conditions (`preCalculateRent :: Batiment -> Bool`)
Aucune.
Post-conditions (`postCalculateRent :: Batiment -> Int -> Bool`)
Retourne le loyer calculé pour le type de bâtiment spécifié.
Description
Calcule le loyer à payer pour un bâtiment résidentiel donné.

`Ville.hs | invariant_creer_imigrant :: Ville -> Bool`

Pré-conditions
Aucune.
Post-conditions
Retourne `True` si les conditions pour créer un immigrant sont remplies, `False` sinon.
Description
Propriété qui vérifie si les conditions nécessaires à l'arrivée d'un immigrant sont bien remplies.

`Shapes.hs | appartient :: Coord -> Forme -> Bool`

Pré-conditions (`preAppartient :: Coord -> Forme -> Bool`)
Les coordonnées doivent être positives.
Post-conditions (`postAppartient :: Coord -> Forme -> Bool -> Bool`)
Si le résultat est vrai, les coordonnées appartiennent à la forme, sinon elles n'y appartiennent pas.
Description
Vérifie si des coordonnées appartiennent bien à une forme donnée.

`Shapes.hs | adjacent :: Batiment -> Int`

Pré-conditions (`preAdjacent :: Coord -> Forme -> Bool`)
Les coordonnées doivent être positives.
Post-conditions (`postAdjacent :: Coord -> Forme -> Bool -> Bool`)
Si le résultat est vrai, les coordonnées sont adjacentes à la forme, sinon elles ne le sont pas.
Description
Vérifie si des coordonnées sont adjacentes à une forme donnée.

`Shapes.hs | formesAdjacentes :: Forme -> Forme -> Bool`

Pré-conditions (`preFormesAdjacentes :: Forme -> Forme -> Bool`)
Aucune.
Post-conditions (`postFormesAdjacentes :: Forme -> Forme -> Bool -> Bool`)
Si le résultat est vrai, les formes sont adjacentes, sinon elles ne le sont pas.
Description
Vérifie si deux formes sont adjacentes.

`Shapes.hs | zonesAdjacentes :: Zone -> Zone -> Bool`

Pré-conditions (`preZonesAdjacentes :: Zone -> Zone -> Bool`)
Aucune.
Post-conditions (`postZonesAdjacentes :: Zone -> Zone -> Bool -> Bool`)
Si le résultat est vrai, les zones sont adjacentes, sinon elles ne le sont pas.
Description
Vérifie si deux zones sont adjacentes.

`Shapes.hs | zoneCollision :: Zone -> Zone -> Bool`

Pré-conditions (`preZoneCollision :: Zone -> Zone -> Bool`)
Aucune.
Post-conditions (`postZoneCollision :: Zone -> Zone -> Bool -> Bool`)
Si le résultat est vrai, les zones se chevauchent, sinon elles ne le font pas.
Description
Vérifie si deux zones entrent en collision (se superposent).

`Shapes.hs | collision :: Forme -> Forme -> Bool`

Pré-conditions (`preCollision :: Forme -> Forme -> Bool`)
Aucune.
Post-conditions (`postCollision :: Forme -> Forme -> Bool -> Bool`)
Si le résultat est vrai, les formes se chevauchent, sinon elles ne le font pas.
Description
Vérifie si deux formes entrent en collision (se superposent).

`Shapes.hs | updateZoneCoord :: Zone -> Coord -> Int -> Int -> Zone`

Pré-conditions (`preUpdateZoneCoord :: Zone -> Coord -> Int -> Int -> Bool`)
Les coordonnées et les dimensions doivent être valides.
Post-conditions (`postUpdateZoneCoord :: Zone -> Coord -> Int -> Int -> Zone -> Bool`)
Vérifie que les nouvelles coordonnées et dimensions sont correctes.
Description
Met à jour les coordonnées d'une zone.

3 Description des Tests Unitaires

On fournit dans cette section une description succincte des tests unitaires utilisés pour tester le code.

3.1 VilleSpec

makeExtraireRoutes

Description : Teste la fonction `extraireRoutes` qui extrait les routes d'une ville.

Cas de test :

- Vérifie que la fonction renvoie les IDs des zones de type `Route` dans une ville donnée.

makeContientRoute

Description : Teste la fonction `contientRoute` qui vérifie si une liste de zones contient une route.

Cas de test :

Vérifie que la fonction renvoie `True` si la liste contient une zone de type `Route`.

Vérifie que la fonction renvoie `False` si la liste ne contient pas de zone de type `Route`.

makeBesideRoad

Description : Teste la fonction `besideRoad` qui vérifie si une zone est à côté d'une route.

Cas de test :

Vérifie que la fonction renvoie `True` si une zone est à côté d'une route.

Vérifie que la fonction renvoie `False` si une zone n'est pas à côté d'une route.

makeGetAdjacent

Description : Teste la fonction `getAdjacent` qui renvoie les zones adjacentes à une zone donnée dans une ville.

Cas de test :

Vérifie que la fonction renvoie les zones adjacentes correctes.

Vérifie que la fonction renvoie une liste vide si aucune zone n'est adjacente.

makePropRoutesConnexes

Description : Teste la fonction `prop_routes_connexes` qui vérifie si les routes sont connexes dans une ville.

Cas de test :

Vérifie que la fonction renvoie `True` si les routes sont connexes.

Vérifie que la fonction renvoie `False` si les routes ne sont pas connexes.

makegetAdjacentRouteIds

Description : Teste la fonction `getAdjacentRouteIds` qui renvoie les IDs des routes adjacentes à une route donnée.

Cas de test :

Vérifie que la fonction renvoie les IDs des routes adjacentes correctes.

makeGraph

Description : Teste la fonction `createGraph` qui crée un graphe des routes dans une ville.

Cas de test :

Vérifie que la fonction crée un graphe correct des routes.

makeIsWithin

Description : Teste la fonction `isWithin` qui vérifie si une forme de bâtiment est à l'intérieur des limites d'une zone.

Cas de test :

Vérifie que la fonction renvoie `True` si la forme de bâtiment est à l'intérieur des limites de la zone.

Vérifie que la fonction renvoie `False` si la forme de bâtiment n'est pas à l'intérieur des limites de la zone.

makeIsOverlapping

Description : Teste la fonction `isOverlapping` qui vérifie si deux formes se chevauchent.

Cas de test :

Vérifie que la fonction renvoie `True` si les formes se chevauchent.

Vérifie que la fonction renvoie `False` si les formes ne se chevauchent pas.

makeIsPowered

Description : Teste la fonction `isPowered` qui vérifie si une zone est alimentée.

Cas de test :

Vérifie que la fonction renvoie `True` si une zone est alimentée par une zone `Wire` adjacente.

Vérifie que la fonction renvoie `True` si une zone est alimentée par une zone `ZE` adjacente.

Vérifie que la fonction renvoie `False` si une zone n'est pas alimentée.

makeCanAddBuilding

Description : Teste la fonction `canAddBuilding` qui vérifie si un bâtiment peut être ajouté à une zone.

Cas de test :

Vérifie que la fonction renvoie `True` si le bâtiment peut être ajouté à l'intérieur des limites de la zone et ne se chevauche pas avec d'autres bâtiments.

Vérifie que la fonction renvoie `False` si le bâtiment chevauche un autre bâtiment dans la zone.

Vérifie que la fonction renvoie `False` si le bâtiment est en dehors des limites de la zone.

makeUpdateCitizens

Description : Teste la fonction `updateCitizens` qui met à jour l'occupation et la position des citoyens en fonction de leur occupation.

Cas de test :

Vérifie que la fonction met à jour la position des citoyens en fonction de leur occupation.

makePollutionZone

Description : Teste la fonction `pollution_zone` qui renvoie le niveau de pollution d'une zone.

Cas de test :

Vérifie que la fonction renvoie le bon niveau de pollution pour les zones industrielles.

Vérifie que la fonction renvoie le bon niveau de pollution pour les zones de production d'énergie.

Vérifie que la fonction renvoie 0 pour les zones non industrielles ou non énergétiques.

makeCoordOfShop

Description : Teste la fonction `coordOfShop` qui renvoie les coordonnées d'un magasin pour un résident.

Cas de test :

Vérifie que la fonction renvoie les coordonnées correctes d'un magasin pour un résident.

makeReachableNodes

Description : Teste la fonction `reachableNodes` qui renvoie l'ensemble des nœuds atteignables à partir d'un nœud donné dans un graphe.

Cas de test :

Vérifie que la fonction renvoie l'ensemble des nœuds atteignables à partir d'un nœud donné dans un graphe.

Vérifie que la fonction renvoie un ensemble vide si le graphe est vide.

makeAStar

Description : Teste la fonction `aStar` qui trouve le chemin le plus court entre deux nœuds connectés dans un graphe.

Cas de test :

Vérifie que la fonction trouve le chemin le plus court entre deux nœuds connectés.

Vérifie que la fonction renvoie `Nothing` s'il n'y a pas de chemin entre deux nœuds.

makeInvariantCreerImigrant

Description : Teste la fonction `invariant_creer_imigrant` qui vérifie que le nombre de zones résidentielles est supérieur à 0 et inférieur ou égal à la somme des zones commerciales et industrielles.

Cas de test :

Vérifie que la fonction renvoie `True` si le nombre de zones résidentielles est supérieur à 0 et inférieur ou égal à la somme des zones commerciales et industrielles.

Vérifie que la fonction renvoie `False` s'il n'y a pas de zones résidentielles.

Vérifie que la fonction renvoie `False` si le nombre de zones résidentielles est supérieur à la somme des zones commerciales et industrielles.

3.2 ShapesSpec

makeZoneForme

Description : Teste la fonction `zoneForme` qui renvoie la forme d'une zone.

Cas de test :

Vérifie que la fonction renvoie la forme correcte pour chaque type de zone.

makeLimites

Description : Teste la fonction `limites` qui renvoie les limites d'une forme.

Cas de test :

Vérifie que la fonction renvoie les limites correctes pour chaque type de forme.

makeAppartient

Description : Teste la fonction `appartient` qui vérifie si une coordonnée appartient à une forme.

Cas de test :

Vérifie que la fonction renvoie `True` si une coordonnée appartient à une forme.

Vérifie que la fonction renvoie `False` si une coordonnée n'appartient pas à une forme.

makeAdjacent

Description : Teste la fonction `adjacent` qui vérifie si une coordonnée est adjacente à une forme.

Cas de test :

Vérifie que la fonction renvoie `True` si une coordonnée est adjacente à une forme.

Vérifie que la fonction renvoie `False` si une coordonnée n'est pas adjacente à une forme.

makeFormesAdjacentes

Description : Teste la fonction `formesAdjacentes` qui vérifie si deux formes sont adjacentes avec une tolérance.

Cas de test :

Vérifie que la fonction renvoie `True` si deux rectangles sont adjacents.

Vérifie que la fonction renvoie `False` si deux rectangles ne sont pas adjacents.

Vérifie que la fonction renvoie `True` si deux rectangles sont adjacents avec une tolérance.

Vérifie que la fonction renvoie `False` si deux rectangles ne sont pas adjacents en dehors de la tolérance.

makeZonesAdjacentes

Description : Teste la fonction `zonesAdjacentes` qui vérifie si deux zones sont adjacentes.

Cas de test :

Vérifie que la fonction renvoie `True` si deux zones ont des formes adjacentes.

Vérifie que la fonction renvoie `False` si deux zones n'ont pas des formes adjacentes.

Vérifie que la fonction renvoie `True` si deux zones ont des formes exactement adjacentes.

Vérifie que la fonction renvoie `False` si deux zones n'ont pas des formes adjacentes en dehors de la tolérance.

makeCollision

Description : Teste la fonction `collision` qui vérifie si deux formes se chevauchent.

Cas de test :

Vérifie que la fonction renvoie `True` si deux rectangles se chevauchent.

Vérifie que la fonction renvoie `False` si deux rectangles ne se chevauchent pas.

makeUpdateZoneCoord

Description : Teste la fonction `updateZoneCoord` qui met à jour les coordonnées et les dimensions d'une zone.

Cas de test :

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone résidentielle (ZR).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone industrielle (ZI).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone commerciale (ZC).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une route (Route).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone d'eau (Eau).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone administrative (Admin).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone de production d'énergie (ZE).

Vérifie que la fonction met à jour correctement les coordonnées et les dimensions d'une zone de fil (Wire).

3.3 Propriétés QuickCheck

Cette section décrit les propriétés QuickCheck définies pour vérifier certaines propriétés du modèle de la ville. Cependant, notez que ces tests ne fonctionnent pas actuellement et nécessitent des ajustements.

prop_isPowered : Vérifie si une zone est alimentée.

```
prop_isPowered :: Ville -> Zone -> Bool
prop_isPowered ville zone = isPowered ville zone == any isWire (getAdjacent ville zone
    )
```

Cette propriété compare le résultat de la fonction `isPowered` avec la condition qu'au moins une des zones adjacentes soit une zone de type `Wire`. Les tests pour cette propriété ne passent pas actuellement.

prop_updateCitizens : Vérifie si la mise à jour des citoyens respecte les invariants.

```
prop_updateCitizens :: Ville -> Bool
prop_updateCitizens ville = all invariantCitoyen (Map.elems (viCit (updateCitizens
    ville)))
```

Cette propriété s'assure que tous les citoyens après la mise à jour respectent les invariants définis. Les tests pour cette propriété ne passent pas actuellement.

prop_invariant_creer_imigrant : Vérifie si l'invariant `creer_imigrant` est respecté.

```
prop_invariant_creer_imigrant :: Ville -> Bool
prop_invariant_creer_imigrant = invariant_creer_imigrant
```

Cette propriété vérifie que l'invariant `creer_imigrant` est respecté pour toute ville donnée. Les tests pour cette propriété ne passent pas actuellement.

3.4 Main de Test

Afin d'exécuter l'ensemble des tests décrits ci-dessus, le code à exécuter en Haskell est le suivant :

```
import Test.Hspec
import ShapesSpec as ShapesS
import VilleSpec as VilleS
main :: IO ()
main = hspec $ do
  ShapesS.shapesSpec
  VilleS.villeSpec
```

Cette description des tests permet de comprendre les différentes fonctions testées et les cas de test spécifiques utilisés pour vérifier leur comportement correct.

```
haskell

import Test.Hspec
import ShapesSpec as ShapesS
import VilleSpec as VilleS

main :: IO ()
main = hspec $ do
  ShapesS.shapesSpec
  VilleS.villeSpec
```

4 Description de l'Implémentation

4.1 Généralités

Afin de faire évoluer sa ville, le joueur a à sa disposition de nombreuses infrastructures qu'il peut construire en dépensant son argent. Ces infrastructures peuvent être construites n'importe où sur la grille, à la condition qu'il n'y ait pas de zone d'eau à cet endroit. De plus, en sélectionnant un élément, le curseur s'adapte et prend une forme et une taille similaire à l'élement que le joueur souhaite placer.

Le panel en haut à droite de l'écran (Figure 2) permet notamment de construire de telles infrastructures :

- Residential $100 : Option pour construire des zones résidentielles, coûtant 100 $.
- Commercial $100 : Option pour construire des zones commerciales, coûtant 100 $.
- Industrial $100 : Option pour construire des zones industrielles, coûtant 100 $.
- Road $10 : Option pour construire des routes, coûtant 10 $.
- Energy $500 : Option pour fournir de l'énergie, coûtant 500 $.
- Wire $5 : Option pour poser des câbles, coûtant 5 $.
- Police Station $100 : Option pour construire un poste de police, coûtant 100 $.

Il met également à disposition des informations primordiales pour maintenir la ville.

- Money : Montant d'argent disponible.
- Population : Nombre de personnes dans la ville.
- Pollution : Niveau de pollution dans la ville.
- Safety : Niveau de sécurité dans la ville.

FIGURE 2 – Panel d'achat des infrastructures, et d'informations sur la ville

4.2 Extensions implémentées

Les extensions implémentées sont les suivantes :

- L'état du jeu (carte, bâtiments, citoyens 'à l'extérieur des bâtiments, animations) doit être affiché 'à l'aide d'une interface graphique (en SDL2), qui représente la zone de jeu, "vue de dessus"
- Les éléments de simulation d´ecrits dans l'ER1 doivent être implémentées
- Les utilisateurs peuvent construire des bâtiments administratifs et des routes, et délimiter des zones R/C/I à la souris et au clavier
- Économie : Des impôts (réglables) sur les sociétés et les citoyens rapportent de l'argent au joueur. Le joueur utilise cet argent pour construire des infrastructures.
- Services : (au moins) une centrale électrique doit être construite par le joueur, et l'électricité doit être acheminée (par contact entre zone, ou par des constructions spéciales) aux différentes zones pour qu'elles se développent. Des commissariats envoient régulièrement des véhicules de patrouille qui se déplacent aléatoirement (ou non) dans la ville.
- Pathfinding : Les citoyens savent comment se déplacer efficacement d'un point à un autre, en évitant les bouchons, si possible.
- Immigration et Émigration : Des places (logements, emplois) disponibles dans une ville attractive attirent les immigrants. A l'inverse, les citoyens trop fatigués, affamés ou sans argent quittent la ville. Une ou plusieurs zones d'interface peuvent être créées pour représenter les points de départ/arrivée des immigrants/émigrants.

4.3 Infrastructures

Parmi les nombreuses infrastructures qui peuvent être construites par le joueur pour améliorer sa ville, il existe : les routes, les zones résidentielles, les zones commerciales, les zones industrielles, les commissariats, les centrales électriques, et les fils.

Les routes (de taille 30x30 sur la grille) sont les briques de base d'une ville. En particulier, les résidences, les industries, les commerces et les commissariats nécessitent d'être desservis et ne peuvent donc être

construits qu'à proximité d'une route. S'il n'en existe pas déjà, une route peut être construite n'importe où. Dans le cas contraire, elle doit être reliée à une autre route.

Les zones résidentielles (70x70) permettent aux citoyens de se reposer pour diminuer leur fatigue. Alimentées, ces zones contiennent 4 bâtiments (cabane ou maison) avec une capacité d'accueil limitée. Un loyer est prélevé aux citoyens qui y résident, permettant notamment au joueur de gagner de l'argent.

La capacité d'accueil et le loyer varient selon le type de bâtiment. Ainsi, une cabane peut accueillir 5 citoyens pour un loyer modeste de 10 unités monétaires, alors qu'une maison peut accueillir 10 citoyens pour un loyer plus élevé de 50 unités. Une zone résidentielle possède initialement des cabanes. Néanmoins, lorsque la sécurité est suffisamment élevée ; que la pollution est suffisamment basse ; que le revenu disponible moyen des résidents est suffisant ; et qu'il y a assez de résidents ; les cabanes deviennent des maisons. Par manque de temps, la fonction de passage de cabanes aux maisons n'a pas pu être implémentée. Néanmoins, tous les pré-requis sont prêts (calcul de la moyenne des loyers, nombre d'habitants dans une zone résidentielle, ...).

Les zones commerciales (70x70), elles, permettent aux citoyens d'acheter de la nourriture et d'augmenter leur jauge de faim. Lorsqu'elles sont alimentées, ces zones contiennent 4 épiceries (30x30) pouvant accueillir jusqu'à 5 citoyens chacune.

Quant aux zones industrielles (70x70), elles permettent aux citoyens de travailler afin de gagner un salaire. Lorsqu'elles sont alimentées, ces zones contiennent 4 ateliers pouvant accueillir jusqu'à 10 ouvriers chacun. Ces zones contribuent à l'augmentation de la pollution de 10 unités si elles sont trop proches (dans un cercle de rayon 500 pixels) des zones résidentielles et/ou des zones commerciales.

Les commissariats (70x70) sont des infrastructures administratives importantes pour assurer la sécurité de la ville. Elles doivent notamment être alimentées en électricité. Lorsqu'un commissariat est actif, un policier patrouille dans les zones résidentielles / commerciales / industrielles alentours. Elles permettent si elle sont proches des zones résidentielles et/ou des zones commerciales d'augmenter le score de sécurité de 1 par commissariat posé.

Enfin, les centrales électriques (70x70) sont primordiales dans une ville car elles permettent d'alimenter les différentes infrastructures de la ville et ainsi les rendre utilisables par les habitants. Elles contribuent par ailleurs à l'augmentation de la pollution de 50 unités. Les centrales électriques ont également la particularité de ne pas être nécessairement placées à proximité d'une route. Pour ce faire, il est nécessaire de les relier aux différentes infrastructures par des fils. Ces derniers ne peuvent être construits qu'à côté d'une centrale électrique ou bien d'un autre fil.

A noter qu'en l'état actuel du projet, l'image des centrales électriques est actuellement visuellement buggée et apparaît comme un losange alors qu'il s'agit bien d'un carré. La zone de collision ne correspond donc pas à ce qui est visible sur la grille.

4.4 Pollution et Sécurité

La pollution est une mesure qui impacte négativement le développement d'une ville. Sa valeur dépend de nombreux facteurs extérieurs. Plus particulièrement, si une centrale électrique ou si une zone industrielle alimentée se situe trop près d'une zone résidentielle ou commerciale alimentée, la pollution augmente de 50 ou 10 unités respectivement.

Concernant la sécurité de la ville, elle peut être améliorée en construisant des zones administratives. Alimentées, elles contiennent des commissariats. Si ces derniers sont à une distance de 500 pixels d'une zone résidentielle ou commerciale, le score de sécurité augmente de 1 unité.

En plus d'évaluer la qualité de vie des citoyens, ces facteurs permettent également de rendre la ville plus attractive et ainsi attirer davantage d'immigrants. Ce sont donc des métriques dont les valeurs dépendent fortement de la population de la ville. Il est donc nécessaire de connaître le nombre d'habitants pour déterminer si leur valeur à un instant donné est plutôt bonne ou mauvaise.

4.5 Habitants

Les habitants de la ville ont chacun un identifiant unique, et se caractérisent par trois mesures différentes : leur argent, leur fatigue et leur faim. Il en existe trois types bien distincts : les immigrants, les citoyens et les émigrants.

Dans les bonnes conditions, les immigrants sont susceptibles d'apparaître à un taux de 1% tous les ticks (entre 1 et 20 immigrants). Pour cela, la pollution doit être suffisamment basse, la sécurité suffisamment élevée, et la population ne doit pas être saturée (au plus 100 immigrants à un instant donné). Un immigrant peut également devenir un citoyen s'il existe un bâtiment résidentiel, un bâtiment industriel et un bâtiment commercial de disponible (cette partie est un peu buggée car il semble que les immigrants peuvent occuper des bâtiments remplis). Les seuils sont calculés dans le bout de code suivant :

```
generateImmigrantsIfCriteriaMet :: GameState -> IO GameState
generateImmigrantsIfCriteriaMet gameState = do
  let numberOfCitizens = getNumberOfVillagers gameState
  let pollutionThreshold = calculatePollutionThreshold numberOfCitizens
  let safetyThreshold = calculateSafetyThreshold numberOfCitizens
  let pollutionRate = getPollutionRate gameState
  let safetyRate = getSafetyRate gameState

  randomValue <- randomRIO (1 :: Int, 100)

  if pollutionRate < pollutionThreshold && safetyRate >= safetyThreshold &&
    invariant_creer_imigrant (ville gameState) && randomValue == 1 &&
    getNumberOfImmigrants gameState < 100
    then do
      let coord = C 0 0
      randomValueIm <- randomRIO (1 :: Int, 20)
      immigrantIds <- generateImmigrantsForZone coord randomValueIm
      let immigrants = map (\cid -> createImmigrant coord cid) immigrantIds
      let updatedGameState = addCitizensToState gameState immigrants
      return updatedGameState
    else return gameState
```

Contrairement aux immigrants, les citoyens sont associés à bâtiment résidentiel où ils peuvent se reposer, à un bâtiment industriel où ils travaillent pour gagner un salaire, et à un bâtiment commercial où ils se nourrissent. Un citoyen devient un émigrant si l'une au moins des trois conditions suivantes est satisfaite : le citoyen n'a plus d'argent ; le citoyen est affamé (faim à 0) ; le citoyen est complètement épuisé (fatigue à 0).

L'état des citoyens est mis à jour tous les 100 ticks. Il s'actualise différement selon l'occupation du citoyen concerné :

```
-- Met a jour la position des citoyens
mettreAJourPosition :: Ville -> Citoyen -> Citoyen
mettreAJourPosition ville citoyen@(Habitant coord (argent, fatigue, faim) batiments
    occupation cid) =
  let citoyenUpdated = if not (isAtHome ville citoyen)
                       then setCitoyenEtat citoyen (argent, max 0 (fatigue - 1), max 0
    (faim - 1))
                       else citoyen
  in case getOccupation citoyenUpdated of
       Move destination ->
         let updatedCitoyen = setCitoyenCoord citoyenUpdated destination
             nextOcc = nextOccupation citoyenUpdated
         in case nextOcc of
              Just occ -> setOccupation updatedCitoyen occ
              Nothing -> updatedCitoyen
       Travailler ->
         if faim < 20
         then setOccupation citoyenUpdated (Move (coordOfShop ville citoyen))
         else if fatigue < 20
         then setOccupation citoyenUpdated (Move (coordOfResidence ville citoyen))
         else
           let newArgent = argent + 200
```

```
                newFatigue = max 0 (fatigue - 10)
                newFaim = max 0 (faim - 5)
            in setCitoyenEtat (setOccupation citoyenUpdated Travailler) (newArgent,
    newFatigue, newFaim)
        Dormir ->
          if fatigue >= 80
          then setOccupation citoyenUpdated (Move (coordOfWork ville citoyen))
          else
            let newFatigue = min 100 (fatigue + 20)
            in setCitoyenEtat (setOccupation citoyenUpdated Dormir) (argent, newFatigue
    , faim)
        Shopping ->
          if faim >= 80
          then setOccupation citoyenUpdated (Move (coordOfWork ville citoyen))
          else
            let newFaim = min 100 (faim + 20)
            in setCitoyenEtat (setOccupation citoyenUpdated Shopping) (argent, fatigue,
     newFaim)
mettreAJourPosition _ citoyen = citoyen

```

Pour résumer (voir Figure 3), lorsqu'un citoyen travaille, il gagne 200 d'argent (salaire), perd 5 de faim et 10 de fatigue. Si l'un de ses attributs (faim ou fatigue) est inférieur à 20, le citoyen arrête de travailler. Il utilise alors l'infrastructure qui lui est associée pour remonter ses attributs jusqu'à la valeur seuil de 80. Par exemple, si le citoyen a faim, alors il part chercher de la nourriture à l'épicerie et gagne 20 de faim jusqu'à atteindre 80. Son comportement est similaire dans le cas de la fatigue, avec les zones résidentielles. Au passage, selon la valeur de ses attributs, cette fonction peut mettre à jour le statut du citoyen en tant qu'émigrant.

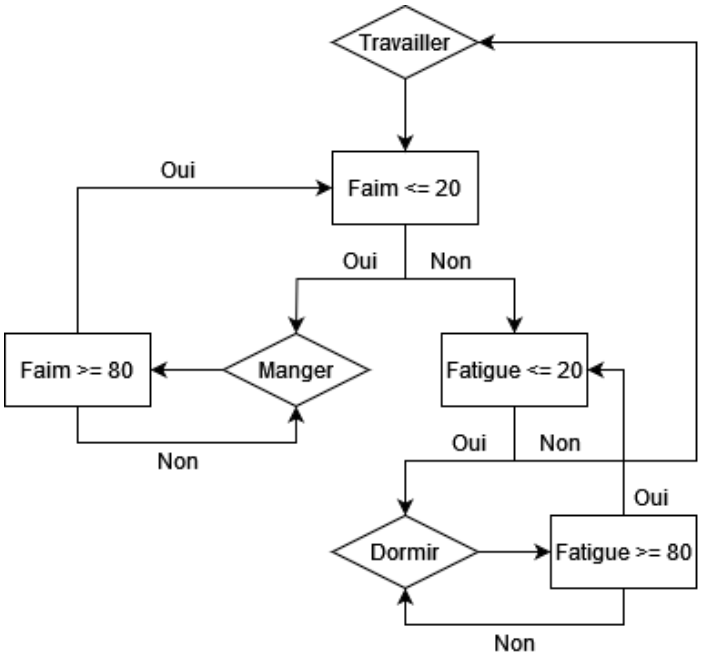

FIGURE 3 – Comportement d'un citoyen

Les émigrants possèdent les mêmes caractéristiques que les citoyens. Cependant, à chaque tick, tous les émigrants sont expulsés de la ville. L'apparition de nouveaux résidents (immigrants) et la disparition d'anciens résidents (émigrants) permettent de dynamiser la population de la ville. Ce cycle nous aura demandé beaucoup de temps et d'efforts à correctement développer.

4.6 Economie

Tous les 100 ticks, tous les habitants doivent payer un loyer : 10 unités pour ceux vivant dans une cabane et 50 pour ceux vivant dans une maison. Ceux qui ne peuvent pas payer deviennent des émigrants et sont expulsés de la ville. Les impôts, réglables, sur les sociétés et les citoyens rapportent de l'argent au joueur. Le joueur utilise cet argent pour construire des infrastructures et développer davantage sa ville.

Conclusion

En conclusion, ce projet aura été une opportunité intéressante pour nous familiariser de façon relativement ludique avec le langage Haskell, en développant un jeu similaire à *Sim City* (Maxis, 1989).

Nous sommes tout particulièrement fiers du cycle immigrant / citoyen / émigrant qui offre une dynamique intéressante au jeu, illustrant le flux constant de la population en fonction de ses capacités à payer le loyer. Par ailleurs, les différentes étapes de la vie d'un habitant, de la recherche de ressources à la gestion de la fatigue et de la faim, enrichissent l'expérience de jeu en ajoutant de la profondeur et du réalisme. Enfin, l'environnement graphique inspiré de Minecraft (Mojang, Microsoft, 2011), en réutilisant des ressources officielles du jeu, ajoute une touche familière et attrayante pour les joueurs.

Néanmoins, la complexité de certains programmes (notamment dans le jeu original) mais aussi des outils à notre disposition (Haskell et SDL) nous aura rendu la tâche difficile. On déplore également le manque d'extensions comme les incendies ou les tsunamis qui auraient pu rendre le jeu plus intéressant. Enfin, d'un point de vue algorithmique, si l'algorithme A^* est implémenté dans notre, il n'est pourtant pas utilisé dans notre modèle car, à l'instar du jeu original, nous avons pris le parti de ne pas surcharger l'écran et la machine du joueur en affichant les résidents. Comme les zones sont adjacentes à des routes, nous avons donc seulement vérifié que les chemins étaient connexes.

En somme, malgré les défis rencontrés, nous pensons que notre projet parvient à offrir une expérience de jeu enrichissante et immersive, capitalisant sur le comportement des citoyens et l'esthétique inspirée de Minecraft. Les limitations actuelles ouvrent, elles, la voie à des améliorations futures. Par exemple, les mesures de pollution et de sécurité pourraient être retravaillées pour être plus indicatives de la qualité de vie des résidents. Pour la sécurité, une idée pourrait ainsi être de considérer quelles sont les zones résidentielles / commerciales protégées par des commissariats, puis calculer le rapport entre le nombre de zones protégées et le nombre de zones total. En normalisant le score de sécurité, il serait ainsi plus facile de le passer à l'échelle.

Bibliographie

[1] IGN, 08 janvier 2008. Sim City: Walkthrough. Consulté pour la première fois le 09 mars 2024.

[2] GameFAQs, 02 mars 1999. Sim City (1989) - FAQ. Consulté pour la première fois le 09 mars 2024.

[3] Wikipédia, 03 janvier 2013. Sim City (1989). Consulté pour la première fois le 17 mai 2024.

[4] Graeme McCutcheon. micropolisJS. Consulté pour la première fois le 17 mai 2024.