Rapport de projet de Programmation Avancée en Fonctionnel

ZEMALI Mohamed et SANTONI Thomas

May 26, 2024

1 Manuel

1.1 Prérequis

Avant de commencer, assurez-vous d'avoir les éléments suivants :

- `stack` installé sur votre machine. Si ce n'est pas le cas, vous pouvez le télécharger et l'installer à partir de https://docs.haskellstack.org/en/stable/install_and_upgrade/.
- Les bibliothèques Haskell suivantes doivent être installées :
  - `base` (>¿= 4.7 ¡5)
  - `containers`
  - `linear`
  - `pqueue`
  - `random`
  - `sdl2`
  - `sdl2-ttf`
  - `text`
- Le code source du projet, organisé dans les fichiers `SimCity.hs`, `Mouse.hs`, `TextureMap.hs`, `SpriteMap.hs`, et `Main.hs`.

1.2 Utilisation du Jeu

Le jeu se joue principalement au clic et aux interactions avec le terminal, ainsi nous avons les fonctionnalités principales suivantes :

- **Placer des zones :** On peut cliquer sur une zone vide, l'invité de commande nous demande si on veut oui ou non placer une zone, on saisit le type de zone qu'on veut construire et ses coordonnées. Les zones sont gratuites, certaines zones nécessitent un entretien et ont donc un coût régulier.
- **Placer des bâtiments :** En cliquant sur une zone, l'invité de commande nous propose de placer des bâtiments. Les bâtiments sont un investissement, ils coûtent donc de l'argent et en rapportent selon les travailleurs/habitants qui y sont. Les bâtiments ne peuvent être placés que dans la zone qui leur correspond.
- **Supprimer des zones/bâtiments :** En faisant clic droit sur une zone ou un bâtiment, on peut afficher ses informations ou le détruire.
- **Tableau d'affichage :** Le jeu possède un tableau d'affichage qui informe le joueur de l'argent actuel, de la répartition immigrant/emigrant/habitant, de la position de la souris, du score de pollution et du score de sécurité.
- **Donner un logement à un immigrant :** En cliquant sur une cabane (normale ou améliorée), on peut choisir d'assigner la résidence à un immigrant. De même pour une épicerie ou un atelier pour donner un travail à un habitant.
- **Placer des câbles :** Placer des câbles est légèrement différent. En effet, le câble que l'on veut placer doit être à proximité d'une centrale électrique (ZE) ou d'un autre câble. De plus, placer un câble près d'une zone améliore tous les bâtiments de la zone, changeant leur sprite et augmentant leur capacité de travailleurs/habitants.

2 Propositions

Dans cette section, nous listons les propositions du projet et ce qu'elles vérifient, classées par type et opérations.

2.1 Zones

1. **propVilleSansCollision :: Ville → Bool :** Cette propriété vérifie deux à deux chaque zone et vérifie qu'elles n'entrent pas en collision. Si une zone est comparée avec elle-même, alors on renvoie True.

2. **propVerifieAllZonesAdjacentesRoute :: Ville → Bool :** Cette propriété vérifie que toutes les zones sont adjacentes à une route. Cela s'applique à toutes les zones à l'exception de Route, Cable, ZE et Eau.

3. **preConstruit :: Ville → Zone → Bool :** Précondition pour construire une zone : cette zone ne doit pas violer la propriété de la ville. Puisqu'on peut construire toute zone, il n'y a pas de condition spécifique sur la zone voulue.

2.2 Bâtiments

1. **propVerifyEntry :: Batiment → Bool :** Cette propriété vérifie que l'entrée d'un bâtiment est adjacente à un bâtiment.

2. **verifyIntLessThanListLength :: Batiment → Bool :** Cette fonction vérifie que dans un bâtiment, il n'y a pas plus d'occupants que la capacité maximum.

3. **prop_entry_appartient_route :: Ville → Bool :** Cette propriété vérifie que l'entrée de chaque bâtiment appartient à une route.

4. **buildingInCorrectZone :: Batiment → Zone → Bool :** Cette fonction vérifie que le bâtiment appartient à la bonne zone. Utile pour ne pas permettre de construire une cabane dans une ZE par exemple.

5. **prop_zoningLaws :: Ville → Bool :** Cette propriété vérifie que chaque bâtiment appartient à la bonne zone en utilisant la fonction `buildingInCorrectZone`.

6. **prop_batiments_in_Zone :: Zone → Bool :** Cette propriété vérifie que chaque bâtiment est bien situé à l'intérieur des limites géométriques de la zone correspondante.

2.3 Occupation

1. **preconditionChangerOccupation :: Citoyen → Occupation → Bool :** Cette précondition vérifie que la nouvelle occupation assignée à un citoyen n'est pas indéfinie, garantissant ainsi que chaque citoyen a une occupation valide et bien définie.

2. **postconditionChangerOccupation :: Citoyen → Occupation → Bool :** Cette postcondition vérifie que, suite au changement d'occupation d'un citoyen, l'occupation mise à jour du citoyen correspond bien à la nouvelle occupation spécifiée.

3. **invariantChangerOccupation :: Citoyen → Occupation → Bool :** Cet invariant vérifie que, lors du changement d'occupation d'un citoyen, les autres attributs du citoyen restent inchangés.

2.4 Ajout de Bâtiments

1. **preconditionAddBuildingToZone :: Int → Batiment → ZoneId → Ville → Bool :** Cette précondition vérifie que l'ajout d'un bâtiment à une zone est valide, en s'assurant que l'utilisateur dispose de suffisamment d'argent, que le prix du bâtiment est non négatif, et que la zone spécifiée existe dans la ville.

2. **postconditionAddBuildingToZone :: Int → Batiment → ZoneId → Ville → (Ville, Int) → Bool :** Cette postcondition vérifie qu'après l'ajout d'un bâtiment à une zone, l'argent de l'utilisateur a été correctement déduit du coût du bâtiment et que le bâtiment a été correctement ajouté à la zone correspondante dans la ville.

3. **invariantAddBuildingToZone :: Int → Batiment → ZoneId → Ville → (Ville, Int) → Bool :** Cet invariant vérifie que le montant d'argent après l'ajout du bâtiment est correct, soit égal à l'argent initial moins le prix du bâtiment, soit inchangé si l'ajout n'a pas eu lieu.

2.5 Ajout d'Immigrants

1. **preconditionAddImmigrantToCabane :: CitId → BatId → Ville → Bool :** Cette précondition vérifie que l'ajout d'un immigrant à une cabane est valide, en s'assurant que l'identifiant du citoyen existe dans la ville et que ce citoyen est bien un immigrant.

2. **postconditionAddImmigrantToCabane :: CitId → BatId → Ville → Bool :** Cette postcondition vérifie qu'après l'ajout d'un immigrant à une cabane, le citoyen a été correctement transformé en habitant avec les caractéristiques mises à jour, et que l'identifiant du nouveau bâtiment correspond bien à celui de la cabane.

3. **invariantAddImmigrantToCabane :: CitId → BatId → Ville → Bool :** Cet invariant vérifie que, suite à l'ajout d'un immigrant à une cabane, le nombre de résidents dans la cabane ne dépasse pas la capacité maximale de celle-ci.

4. **preconditionAddImmigrants :: Int → Ville → CitId → Bool :** Cette précondition vérifie qu'il n'y a actuellement aucun immigrant dans la ville avant d'ajouter de nouveaux immigrants.

5. **postconditionAddImmigrants :: Int → Ville → CitId → Ville → CitId → Bool :** Cette postcondition vérifie qu'après l'ajout d'un nombre spécifié d'immigrants à la ville, le nombre d'immigrants est correct, que les identifiants des nouveaux immigrants correspondent aux identifiants attendus, et que l'identifiant du dernier nouvel immigrant est correct.

2.6 Mise à jour du Travail

1. **preconditionUpdateTravail :: Ville → Citoyen → Bool :** Cette précondition vérifie que le citoyen est un habitant et qu'il a un travail défini avant de mettre à jour son statut de travail.

2. **postconditionUpdateTravail :: Ville → Citoyen → Bool :** Cette postcondition vérifie qu'après la mise à jour du travail d'un citoyen, les changements dans ses attributs (tels que la coordination, l'argent, la fatigue et la faim) sont corrects en fonction de son occupation actuelle. Par exemple, si le citoyen dort, sa fatigue doit augmenter.

3. **invariantUpdateTravail :: Citoyen → Bool :** Cet invariant vérifie que, suite à la mise à jour du travail d'un citoyen, ses coordonnées, son logement et ses attributs biologiques (tels que la fatigue, la faim et l'argent) restent inchangés, garantissant ainsi que seules les occupations et leurs effets sont modifiés.

3 Tests

Dans cette section, nous décrivons les différents tests que nous avons mis en place pour vérifier le bon fonctionnement des fonctions principales de notre projet.

3.1 Tests pour la Fonction `addZone`

Nous avons vérifié que la fonction `addZone` respecte les préconditions et postconditions suivantes :

- Ajoute une nouvelle zone si les préconditions et postconditions sont respectées.
- Ne pas ajouter une nouvelle zone si la précondition échoue.
- Traçage d'un message si la postcondition échoue.

3.2 Tests pour la Fonction `addImmigrants`

Les tests de la fonction `addImmigrants` vérifient que :

- Les immigrants sont ajoutés correctement lorsqu'il n'y a pas d'immigrants existants.
- Les nouveaux immigrants ont les bonnes coordonnées, statistiques et occupations.
- Les identifiants des nouveaux immigrants sont corrects.
- La ville et l'identifiant retournés sont corrects lorsqu'il y a déjà des immigrants existants.

3.3 Tests pour la Fonction `augmenterCapaciteBatiments`

Les tests pour `augmenterCapaciteBatiments` vérifient que :

- La capacité des cabanes est augmentée de 10.
- La capacité des ateliers est augmentée de 4.
- La capacité des épiceries est augmentée de 4.
- Les autres types de bâtiments ne sont pas modifiés.

3.4 Tests pour les Fonctions `cityCost` et `pollutionScore`

Les tests pour `cityCost` vérifient que :

- Le coût est calculé correctement pour une ville contenant différents types de zones.
- Le coût est 0 pour une ville sans zones spécifiques.

Les tests pour `pollutionScore` vérifient que :

- Le score de pollution est calculé correctement pour une ville contenant différents types de zones.
- Le score de pollution est 0 pour une ville sans zones spécifiques affectant la pollution.

3.5 Tests pour la Fonction `verifieAdjacenceAuneRoute`

Les tests pour `verifieAdjacenceAuneRoute` vérifient que :

- Retourne `True` si la zone est adjacente à une route.
- Retourne `False` si la zone n'est pas adjacente à une route.

3.6 Tests pour la Fonction `collision2Zones`

Les tests pour `collision2Zones` vérifient que :

- Retourne `True` si deux zones ne se chevauchent pas.
- Retourne `False` si deux zones se chevauchent.

3.7 Tests pour la Fonction `verifyIntLessThanListLength`

Les tests pour `verifyIntLessThanListLength` vérifient que :

- Retourne `True` si le nombre de citoyens dans un bâtiment est inférieur ou égal à la capacité du bâtiment.
- Retourne `False` si le nombre de citoyens dans un bâtiment est supérieur à la capacité du bâtiment.

3.8 Tests pour la Fonction `buildingInCorrectZone`

Les tests pour `buildingInCorrectZone` vérifient que :

- Retourne `True` si un bâtiment est dans la zone correcte.
- Retourne `False` si un bâtiment n'est pas dans la zone correcte.

3.9 Tests pour la Fonction `checkAndTransformCitizens`

Les tests pour `checkAndTransformCitizens` vérifient que :

- Les citoyens ayant faim ou sommeil inférieur ou égal à 0 sont transformés en émigrants.
- Les autres citoyens restent inchangés.

3.10 Tests pour la Fonction `safetyScore`

Nous avons défini les préconditions, postconditions et invariants pour la fonction `safetyScore` et nous avons mis en place les tests suivants :

- **Précondition** : La ville doit être correctement initialisée avec des zones et des citoyens. La ville ne doit pas être vide.
- **Postcondition** : Le score de sécurité doit être un entier, qui peut être positif ou négatif en fonction du nombre de bâtiments près des commissariats et du nombre de citoyens sans emploi.
- **Invariant** : Les valeurs utilisées pour le calcul (à savoir `buildingsNearCommissariatsCount` et `unemployedCount`) doivent être des entiers non négatifs.

Les tests vérifient que :

- La fonction retourne un score correct pour une ville avec un commissariat et aucun chômeur.
- La fonction retourne un score correct pour une ville avec un commissariat et des chômeurs.
- La fonction retourne un score de sécurité négatif pour une ville avec plusieurs chômeurs.
- La fonction retourne un score de sécurité positif pour une ville avec plusieurs commissariats et aucun chômeur.
- La fonction respecte la précondition pour une ville correctement initialisée.
- La fonction respecte la postcondition pour une ville donnée.
- La fonction respecte l'invariant pour une ville donnée.

Voici les tests détaillés pour `safetyScore` :

Listing 1: Tests pour `safetyScore`

```
import Test.Hspec
import SimCity

main :: IO ()
main = hspec $ do
    describe "safetyScore" $ testSafetyScore

testSafetyScore :: Spec
testSafetyScore = describe "safetyScore" $ do
    it "devrait retourner un score correct pour une ville avec un commissariat et aucun
        let ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
        safetyScore ville `shouldBe` 10
```

```
it "devrait retourner un score correct pour une ville avec un commissariat et des ch
    let citoyen1 = Habitant (C 1 1) (0, 0, 0) (BatId 1, Nothing, Nothing) Chomage
        ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
    safetyScore ville `shouldBe` -10

it "devrait retourner un score de s curit n gatif pour une ville avec plusieurs
    let citoyen1 = Habitant (C 1 1) (0, 0, 0) (BatId 1, Nothing, Nothing) Chomage
        citoyen2 = Habitant (C 2 2) (0, 0, 0) (BatId 1, Nothing, Nothing) Chomage
        ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
    safetyScore ville `shouldBe` -30

it "devrait retourner un score de s curit positif pour une ville avec plusieurs c
    let ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
    safetyScore ville `shouldBe` 20

it "devrait respecter la pr condition pour une ville correctement initialis e" $ d
    let ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
    preconditionSafetyScore ville `shouldBe` True

it "devrait respecter la postcondition pour une ville donn e" $ do
    let citoyen1 = Habitant (C 1 1) (0, 0, 0) (BatId 1, Nothing, Nothing) Chomage
        ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
        score = safetyScore ville
    postconditionSafetyScore ville score `shouldBe` True

it "devrait respecter l'invariant pour une ville donn e" $ do
    let citoyen1 = Habitant (C 1 1) (0, 0, 0) (BatId 1, Nothing, Nothing) Chomage
        ville = V (Map.fromList [(ZoneId 0, Admin (Rectangle (C 0 0) 10 10) (Commiss
    invariantSafetyScore ville `shouldBe` True
```

Ces tests permettent de garantir que les différentes fonctions de notre projet fonctionnent correctement et respectent les invariants et conditions spécifiés.

3.11 Générateurs

Dans le cadre de ce projet, nous avons tenté de développer divers générateurs pour faciliter la création de données de test. Ces générateurs étaient destinés à produire des instances de villes, zones, bâtiments et citoyens de manière aléatoire, tout en respectant certaines contraintes et invariants définis. Cependant, nous avons rencontré plusieurs problèmes lors de leur implémentation. Nous nous sommes inspirés du TME effectué dans le cadre de l'UE.

- **Coordonnées et Formes** : Nous avons implémenté des générateurs pour les coordonnées (`genCoord`) et différentes formes géométriques telles que les segments horizontaux (`genHSegment`), les segments verticaux (`genVSegment`) et les rectangles (`genRectangle`).
- **Zones** : Le générateur de zones (`genZone`) avait pour but de créer des zones résidentielles (ZR), industrielles (ZI), commerciales (ZC) et administratives (Admin), en plaçant aléatoirement des bâtiments à l'intérieur de ces zones sans chevauchement.
- **Bâtiments** : Les générateurs de bâtiments (`genCabane`, `genAtelier`, `genEpicerie`, `genCommissariat`) visaient à produire différents types de bâtiments avec des formes et des coordonnées aléatoires, et à les insérer dans les zones correspondantes.
- **Citoyens** : Les générateurs de citoyens (`genImmigrant`, `genHabitant`, `genEmigrant`) avaient pour objectif de créer des citoyens avec des états et des occupations aléatoires, et de les assigner à des logements, des entreprises ou des magasins en fonction de leurs rôles.
- **Villes** : Le générateur de villes (`genVille`) cherchait à composer des villes en générant et en ajoutant des zones et des citoyens de manière cohérente, en s'assurant que toutes les zones

respectent les invariants définis, tels que l'absence de collision entre zones (`collision2Zones`) et l'adjacence des zones à une route (`verifieAdjacenceAuneRoute`).

- **Invariants** : Les propriétés comme `property_inv_genVille` étaient destinées à vérifier que les villes générées respectaient les invariants nécessaires à la cohérence du jeu.

Ces générateurs devaient être essentiels pour automatiser les tests et garantir que notre implémentation gère correctement une variété de scénarios possibles. Ils avaient également pour but de créer des cas de test réalistes et variés, améliorant ainsi la robustesse et la fiabilité du système.

Voici un extrait de code montrant l'utilisation de certains de ces générateurs :

```
genVille :: Int -> Int -> Gen Ville
genVille maxX maxY = do
    initialCabane <- genCabane maxX maxY (ZR (Rectangle (C 0 0) maxX maxY) [])
    let initialZone = case initialCabane of
            Just cabane -> updateZoneWithBuilding cabane (ZR (Rectangle (C 0 0) maxX maxY) [])
            Nothing -> ZR (Rectangle (C 0 0) maxX maxY) []
    zones <- generateZones initialZone 10
    return $ V (Map.fromList (zip (map ZoneId [0..]) zones)) Map.empty
```

Malgré nos efforts pour développer ces générateurs, nous avons rencontré plusieurs problèmes techniques qui ont limité leur efficacité. Néanmoins, ces outils nous ont permis de tester certaines parties de notre code et d'assurer un certain niveau de robustesse.

4 Extensions réalisées

4.1 L'économie du jeu

Le joueur possède un montant d'argent au départ. Tous les n tours, celui-ci reçoit un montant fixe qui fait penser à une subvention ou quand on fait un tour au Monopoly. Les bâtiments coûtent de l'argent et diminuent donc celui du joueur, mais si des travailleurs s'y trouvent ou des habitants dans des cabanes, alors le joueur reçoit de l'argent à la manière d'impôts ou de loyers. L'argent peut également diminuer de manière régulière. En effet, si l'utilisateur possède une ZE (Zone Électrique), des câbles ou des points d'eau, alors tous les n ticks, de l'argent lui est déduit en guise d'entretien.

La fonction suivante calcule le coût de maintien de la ville :

```
cityCost :: Ville -> Int
cityCost ville =
    Map.foldr step 0 (viZones ville)
    where
        step (ZE _) acc = acc + 2000 -- la ZE coute 2000 a entretenir
        step (Cable _) acc = 200 + acc -- les cables aussi coutent de l'argent
        step (Admin _ _) acc = acc + 1000 -- la police coute aussi
        step (Eau _) acc = acc + 300 -- l'eau aussi, on entretient pour garder un bon sc
        step _ acc = acc -- le reste est considere comme un investissement
```

Le système d'impôt de notre jeu permet au joueur de générer des revenus à partir des bâtiments et des zones. La fonction principale `calculateMoney` calcule l'argent total que le joueur reçoit en additionnant l'argent généré par chaque zone de la ville. Pour chaque zone, la fonction `moneyFromZone` somme les revenus de tous les bâtiments présents dans cette zone. Les bâtiments génèrent des revenus en fonction de leur type et du nombre de travailleurs ou d'habitants qu'ils contiennent. Par exemple, une épicerie et un atelier génèrent chacun 30 unités d'argent par travailleur, tandis qu'une cabane génère 20 unités d'argent par habitant. Les autres types de bâtiments ne génèrent pas de revenus.

```
calculateMoney :: Ville -> Int
calculateMoney ville = foldr (+) 0 $ map moneyFromZone (Map.elems $ viZones ville)

-- Calculer l'argent d'une zone specifique
moneyFromZone :: Zone -> Int
```

```
moneyFromZone zone = sum $ map moneyFromBuilding (buildingsFromZone zone)

moneyFromBuilding :: Batiment -> Int
moneyFromBuilding (Epicerie (Rectangle w h) workers) = 30 * length workers
moneyFromBuilding (Cabane n) = 20 * n
moneyFromBuilding (Atelier (Rectangle w h) workers) = 30 * length workers
moneyFromBuilding _ = 0 -- les autres ne rapportent rien
```

Le joueur peut voir tout son argent dans la fenêtre graphique.

4.2 Service

Le système de mise à jour avec l'électricité dans notre jeu permet d'augmenter la capacité des bâtiments situés à proximité des câbles électriques. La fonction `updateWithElectricity` parcourt toutes les zones de la ville, identifie les câbles, et met à jour les zones adjacentes en augmentant la capacité des bâtiments qu'elles contiennent. Les câbles jouent un rôle essentiel en fournissant de l'électricité, ce qui améliore les bâtiments résidentiels, industriels et commerciaux.

La fonction `addCable` permet d'ajouter un câble à la ville. Elle vérifie d'abord si le nouveau câble peut être connecté à une zone électrique existante ou à un autre câble. Si c'est le cas, le câble est ajouté à la ville et la fonction `updateWithElectricity` est appelée pour mettre à jour les capacités des bâtiments adjacents.

```
augmenterCapaciteBatiments :: [Batiment] -> [Batiment]
augmenterCapaciteBatiments = map augmenterCapacite
  where
    augmenterCapacite (Cabane forme coord capacite residents batId) = Cabane forme coord
    augmenterCapacite (Atelier forme coord capacite residents batId) = Atelier forme coo
    augmenterCapacite (Epicerie forme coord capacite residents batId) = Epicerie forme c
    augmenterCapacite autre = autre

-- Fonction pour verifier si une zone est adjacente a une autre zone
validAdjacentZone :: Forme -> Zone -> Bool
validAdjacentZone newCableForm zone = case zone of
    ZE forme -> adjacentes forme newCableForm
    Cable forme -> adjacentes forme newCableForm
    _ -> False

-- Mettre a jour la ville avec l'electricite en augmentant la capacite des batiments
updateWithElectricity :: Ville -> Ville
updateWithElectricity ville =
    let zones = Map.toList (viZones ville)
        cables = filter (isCable . snd) zones
        updatedZones = foldl updateZoneElectricity (viZones ville) cables
    in ville { viZones = updatedZones }
  where
    isCable (Cable _) = True
    isCable _ = False

    updateZoneElectricity :: Map.Map ZoneId Zone -> (ZoneId, Zone) -> Map.Map ZoneId Zon
    updateZoneElectricity zones (zoneId, Cable forme) =
        let adjacents = filter (adjacentes forme . zoneForme . snd) (Map.toList zones)
        in foldl augmenterCapaciteSiAdjacente zones adjacents

    augmenterCapaciteSiAdjacente :: Map.Map ZoneId Zone -> (ZoneId, Zone) -> Map.Map Zon
    augmenterCapaciteSiAdjacente zones (adjZoneId, ZR forme batiments) =
        Map.adjust (\(ZR forme _) -> ZR forme (augmenterCapaciteBatiments batiments)) ad
    augmenterCapaciteSiAdjacente zones (adjZoneId, ZI forme batiments) =
```

```
        Map.adjust (\(ZI forme _) -> ZI forme (augmenterCapaciteBatiments batiments)) ad
    augmenterCapaciteSiAdjacente zones (adjZoneId, ZC forme batiments) =
        Map.adjust (\(ZC forme _) -> ZC forme (augmenterCapaciteBatiments batiments)) ad
    augmenterCapaciteSiAdjacente zones _ = zones

-- Fonction pour ajouter un cable a la ville
addCable :: Zone -> Ville -> Ville
addCable zoneCable ville =
    let zones = Map.elems (viZones ville)
        validAdjacentZone zone = case zone of
            ZE forme -> adjacentes forme (zoneForme zoneCable)
            Cable forme -> adjacentes forme (zoneForme zoneCable)
            _ -> False
        isValidCable = any validAdjacentZone zones
    in if isValidCable
       then
           let newZone = Cable (zoneForme zoneCable)
               updatedVille = construit ville newZone
           in (updateWithElectricity updatedVille)
       else ville
```

4.3 Qualité de vie

Dans notre implémentation, la qualité de vie se définit par la sûreté de la ville, c'est-à-dire le nombre d'habitations couvertes par un commissariat. Cette sûreté baisse (ce qui augmente le score de qualité de vie) avec le nombre de chômeurs. La fonction `safetyScore` calcule ce score en prenant en compte ces deux facteurs :

```
safetyScore :: Ville -> Int
safetyScore ville =
    let unemployedCount = countUnemployedCitizens ville
        buildingsNearCommissariatsCount = countBuildingsNearCommissariats ville
    in buildingsNearCommissariatsCount * 10 - unemployedCount * 20
```

La fonction `safetyScore` calcule le score de sûreté de la ville en utilisant deux sous-fonctions : `countUnemployedCitizens` et `countBuildingsNearCommissariats`. Le score est obtenu en multipliant le nombre de bâtiments proches des commissariats par 10 et en soustrayant le produit du nombre de chômeurs par 20.

```
countUnemployedCitizens :: Ville -> Int
countUnemployedCitizens ville =
    length $ filter isUnemployed (Map.elems (viCit ville))
  where
    isUnemployed (Habitant _ _ _ Chomage) = True
    isUnemployed _ = False
```

La fonction `countUnemployedCitizens` compte le nombre de citoyens sans emploi dans la ville. Elle parcourt la liste des citoyens et utilise un filtre pour sélectionner ceux dont l'occupation est `Chomage`.

```
countBuildingsNearCommissariats :: Ville -> Int
countBuildingsNearCommissariats ville =
    length $ filter isNearCommissariat (getAllBuildings ville)
  where
    commissariats = getCommissariats ville
    isNearCommissariat building = any (\(Commissariat coord _ _) -> distance (getEntry b

    getCommissariats :: Ville -> [Batiment]
    getCommissariats ville = filter isCommissariat (getAllBuildings ville)
```

```
isCommissariat (Commissariat _ _ _) = True
isCommissariat _ = False
```

La fonction `countBuildingsNearCommissariats` compte le nombre de bâtiments situés à proximité d'un commissariat. Elle utilise une liste de commissariats obtenue par la fonction `getCommissariats` et vérifie pour chaque bâtiment s'il se trouve à une distance de 200 unités ou moins d'un commissariat. Les fonctions `isNearCommissariat` et `getCommissariats` aident à identifier les bâtiments proches des commissariats et à filtrer les commissariats respectivement.

En combinant ces deux comptages, la fonction `safetyScore` fournit une mesure de la qualité de vie dans la ville, tenant compte de la couverture des services de sécurité et du taux de chômage/d'Emigrant.

4.4 Pathfinding

Pour ce projet, l'un des objectifs était de développer un système de pathfinding permettant aux citoyens de se déplacer efficacement d'un point à un autre, en évitant les bouchons si possible. Malgré nos efforts et l'utilisation de ChatGPT pour obtenir de l'aide, nous n'avons pas réussi à implémenter cette fonctionnalité de manière satisfaisante. Des traces de nos essais et des tentatives de développement de cette fonctionnalité sont néanmoins présentes dans le code du projet, nous avons essayé d'utiliser A*.

4.5 Immigration

Une des fonctionnalités importantes de notre jeu consiste à ajouter des logements pour les immigrants et à les transformer en citoyens actifs. La fonction `addImmigrantToCabane` permet de réaliser cette tâche en vérifiant d'abord que l'immigrant existe et que la cabane cible a une capacité suffisante pour accueillir de nouveaux résidents. Si ces conditions sont remplies, l'immigrant est transformé en habitant et est ajouté à la liste des résidents de la cabane.

```
addImmigrantToCabane :: CitId -> BatId -> Ville -> Maybe Ville
addImmigrantToCabane citId batId ville =
    case Map.lookup citId (viCit ville) of
        Just (Immigrant coord info occupation) ->
            case findCabane batId ville of
                Just (Cabane forme cabCoord capacite residents batId) | length residents
                    let newResidents = residents ++ [citId]
                        updatedCabane = Cabane forme cabCoord capacite newResidents batI
                        newCitoyen = Habitant cabCoord (400,400,400) (batId, Nothing, No
                        updatedCitoyens = Map.insert citId newCitoyen (viCit ville)
                        newVille = updateVilleWithBuilding updatedCabane ville
                    in Just newVille { viCit = updatedCitoyens }
                _ -> trace "Cabane is full or not found" Nothing
        _ -> trace "Immigrant not found" Nothing
```

Cette fonction suit les étapes suivantes :

1. Vérifier si l'immigrant avec l'identifiant `citId` existe dans la ville.
2. Trouver la cabane avec l'identifiant `batId` et vérifier si elle a la capacité d'accueillir un nouvel immigrant.
3. Si la cabane peut accueillir l'immigrant, ajouter l'immigrant à la liste des résidents de la cabane et le transformer en habitant.
4. Mettre à jour la liste des citoyens de la ville avec le nouvel habitant.
5. Retourner la nouvelle version de la ville avec les mises à jour appropriées.

4.6 Transformation des citoyens en émigrants

Dans notre jeu, il est crucial de surveiller l'état des citoyens pour maintenir une qualité de vie acceptable. Pour cela, nous avons implémenté une fonctionnalité qui transforme automatiquement les citoyens en émigrants si leur faim ou leur sommeil atteint zéro. Cette transformation reflète la situation où des citoyens quittent la ville en raison de conditions de vie insatisfaisantes.

La fonction `checkAndTransformCitizens` parcourt tous les citoyens de la ville et vérifie leur état de faim et de sommeil. Si l'un de ces attributs est à zéro, le citoyen est transformé en émigrant. Voici la définition de cette fonction :

```
-- Fonction pour transformer les citoyens en emigrants si leur faim ou leur sommeil est
checkAndTransformCitizens :: Ville -> Ville
checkAndTransformCitizens ville =
    ville { viCit = Map.map checkCitizen (viCit ville) }
  where
    checkCitizen :: Citoyen -> Citoyen
    checkCitizen citoyen@(Immigrant _ (faim, _, sommeil) _)
        | faim <= 0 || sommeil <= 0 = transformToEmigrant citoyen
        | otherwise = citoyen
    checkCitizen citoyen@(Habitant _ (faim, _, sommeil) _ _)
        | faim <= 0 || sommeil <= 0 = transformToEmigrant citoyen
        | otherwise = citoyen
    checkCitizen citoyen = citoyen  -- Ne rien faire pour les autres types de citoyens

    transformToEmigrant :: Citoyen -> Citoyen
    transformToEmigrant (Immigrant coord _ _) = Emigrant coord Chomage
    transformToEmigrant (Habitant coord _ _ _) = Emigrant coord Chomage
    transformToEmigrant citoyen = citoyen  -- Juste au cas ou, meme si cela ne devrait p
```

Cette fonction suit plusieurs étapes :

1. **Parcours des citoyens** : Utilisation de `Map.map` pour appliquer une vérification à chaque citoyen dans la ville.
2. **Vérification des citoyens** : La fonction `checkCitizen` vérifie si la faim ou le sommeil du citoyen est à zéro. Si c'est le cas, le citoyen est transformé en émigrant.
3. **Transformation en émigrant** : La fonction `transformToEmigrant` prend un citoyen (immigrant ou habitant) et le transforme en émigrant en conservant ses coordonnées et en lui assignant l'occupation `Chomage`.

En implémentant cette fonctionnalité, nous assurons une gestion dynamique de la population de la ville, où les citoyens mécontents ou négligés quittent la ville, reflétant un aspect réaliste de la gestion urbaine.

5 Difficultés et réussite :

L'aspect graphique était extrêmement compliqué, de plus nous avons choisi d'utiliser les clics de la souris donc des marges d'erreurs ont du être adapté. Pour l'affichage nous avons utilisé l'aide de ChatGPT, mais aucun github ou utilisation de copilot n'a été faite pour tout le reste du projet (sauf A*). Le plus compliqué du projet reste tout de même l'aspect création de jeu, en effet nous trouvons le Haskell peu générique et nous n'en avons pas fait une utilisation efficace, faire du pattern matching a chaque type rencontré devient vite fatiguant, quand en Java on a des interfaces ou Super-classe le faisant efficacement. Le Path finding a été très chronophage, nous y avons passé 1 journée pour peu voire pas de résultat malgré nos effort. Nous avons tout de même produit un jeu fonctionnel malgré ces difficultés, le roulement marche donc un citoyen va à son travail, puis si il a faim va se nourrir ou dormir. Nous avons donc mis en place les services d'améliorations avec la centrale, un système de monnaie fonctionnel et de score de sécurité/pollution. Ce projet nous a prouvé la difficulté et la précision de programmer en fonctionnel, notamment sur un jeu vidéo qui nécéssite énormément de

check. Le jeu n'est finalemend pas très bon niveau gameplay car la souris est trop difficile à manier, malgré la présence de fonctionnalités. Les sprites viennent de générateur d'image par IA ou de Galaxy Life (le jeu mobile).

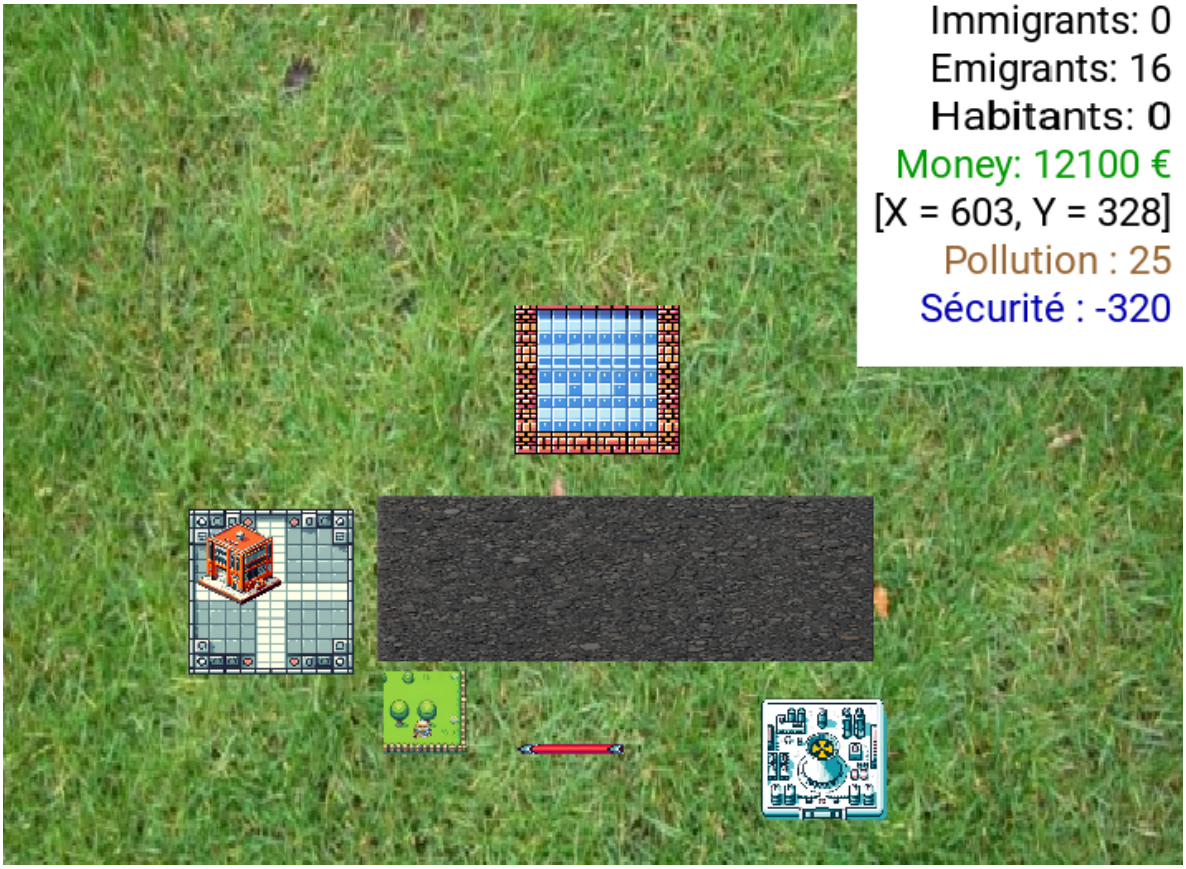

Figure 1: Exemple d'une partie d'un joueur AFK/Pas très bon

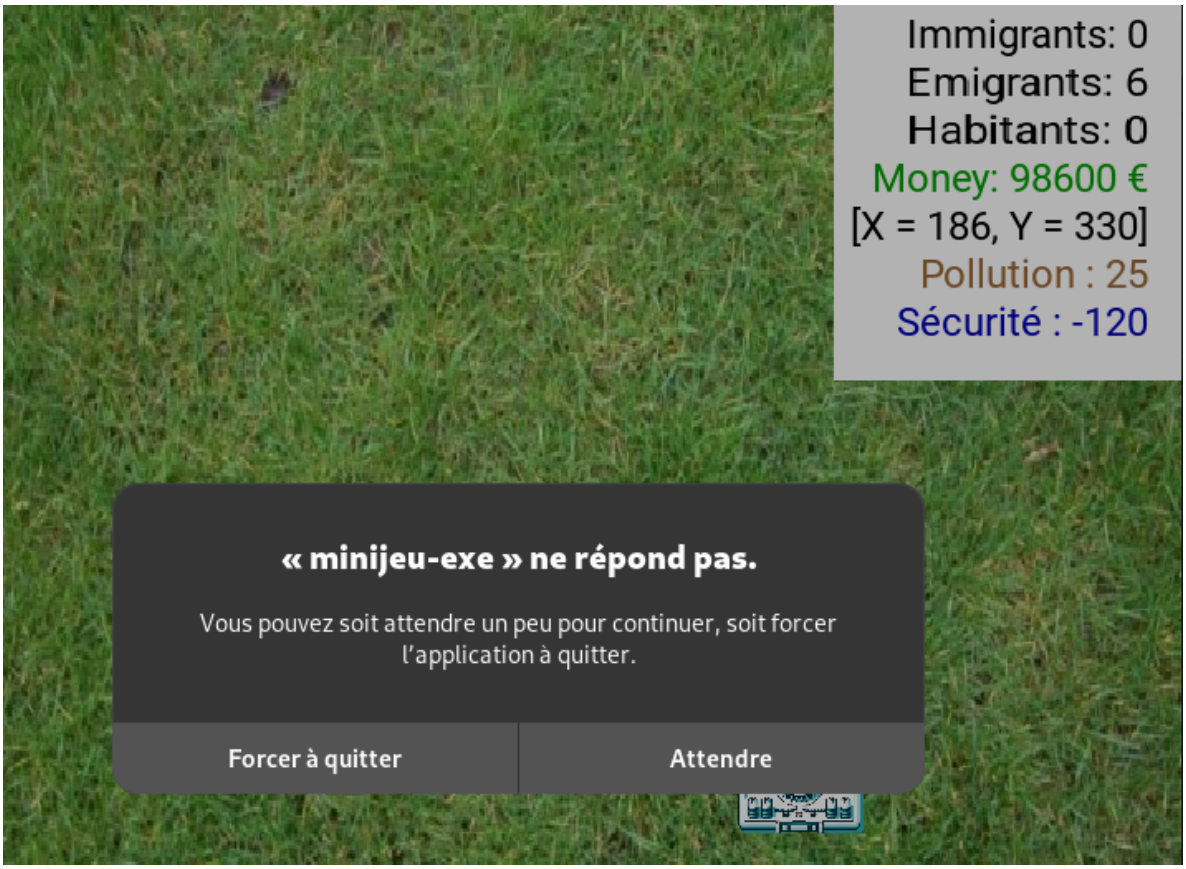

Figure 2: Problème lié à SDL