ANTONUCCI Rémi
CHEN Jacky

EISE4
27/01/2023

Projet CPP

Rapport du jeu « Qatar City Simulator »

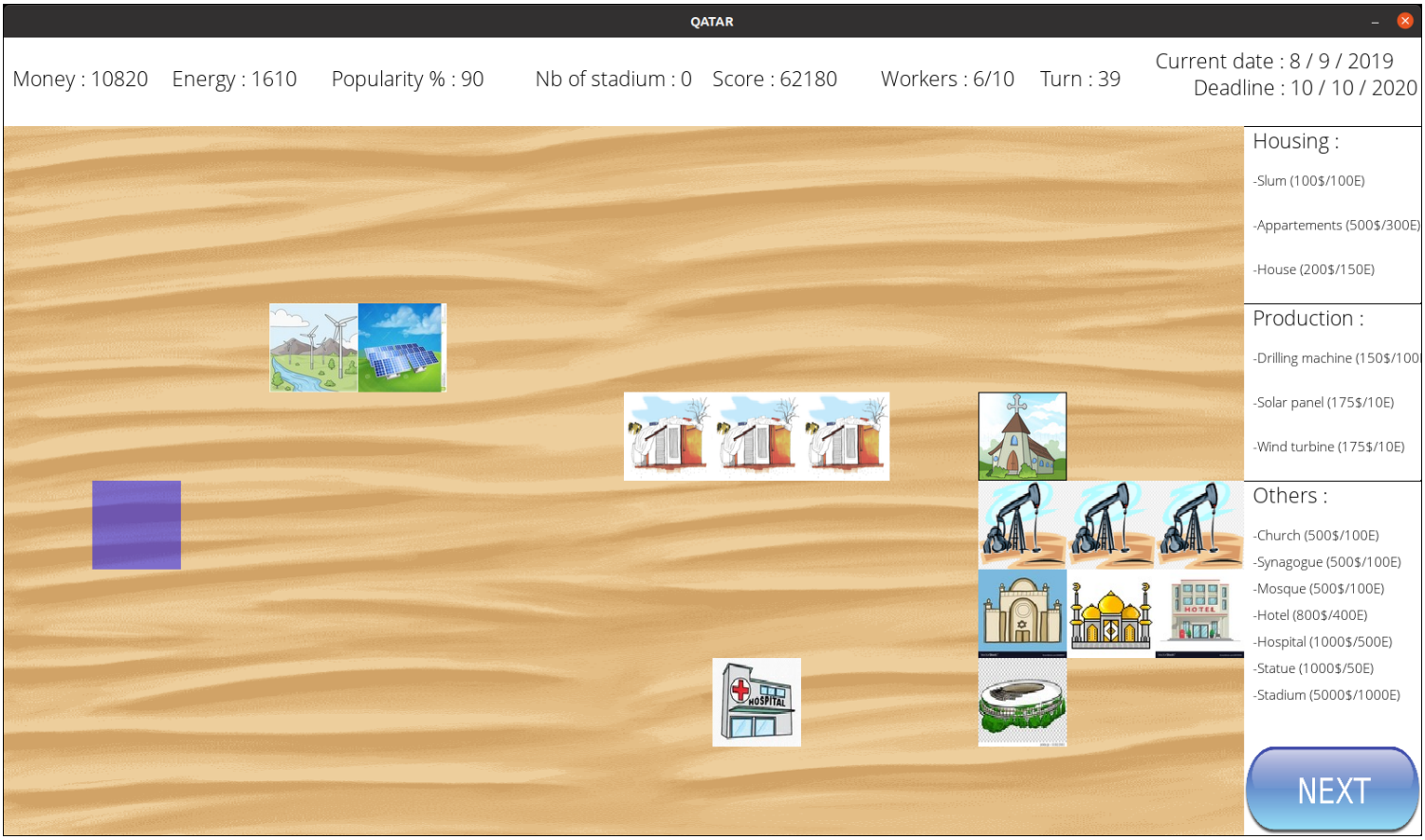


Table des matières

Présentation de notre jeu :

Le jeu « Qatar City Simulator » s'inscrit dans le genre des jeux de gestion. Il s'inspire de certains jeux connus tels que Clash Of Clans, Civilisation, ou encore Sims City. Le thème imposé était « coupe du monde ». On a donc décidé de reprendre les évènements récents en créant un jeu où on incarne des dirigeants du Qatar (pays organisateur de la coupe du monde de football 2022) . Le but du jeu est de construire 5 stade de football avant une certaine deadline.

Pour arriver à cet objectif, le joueur doit bien gérer ses ressources qui sont :

- l'argent de son pays
- l'énergie disponible
- la disponibilité ou non des ses travailleurs

Cela s'organise en tours. Chaque tour le joueur peut construire autant de bâtiments qu'il veut dans la limite de ses ressources disponibles. Quand la construction d'un bâtiment est lancée, le joueur perd définitivement de l'argent et de l'énergie en fonction de ce qu'il décide de construire, et un travailleur ne sera plus disponible le temps que le bâtiment soit prêt.
Le temps de construction se définit en nombre de tours.

Il y a différents types de bâtiments :

- Les « HousingBuilding » : permettent de loger les travailleurs. En construisant ce type de bâtiment on gagne des travailleurs, et on peut donc plus construire au prochain tour.
- Les « ProductionBuilding » : servent à rapporter de l'énergie et de l'argent à chaque tour.
- Les autres bâtiments servent à rapporter du score ou de la popularité. Plus on a de popularité plus on gagne automatiquement de l'argent par tour.

Détaillons les particularités de certains bâtiments.

- Les Stadiums sont la condition de victoire. Il en faut 5.
- Les « ReligiousBuilding » apportent un bonus de score si les 3 principales religions sont représentées (christianisme, judaïsme, islam).
- Les hôtels rapportent encore plus d'argent en fonction de la popularité.
- Les statues rapportent juste de la popularité.

Les hôpitaux ont une fonction à part, un peu « décalée » de l'objectif principal.

En effet, on a essayé d'introduire une dimension éthique, en incluant la pollution et le bien-être des travailleurs. Ceci se fait de façon très simple : si les travailleurs sont logés dans des bidonvilles ou si le joueur utilise des machines de forage, des travailleurs meurent.
Pour la pollution on a créé une émission de CO_2 pour chaque bâtiment, mais la machine de forage pollue énormément, malgré que ça soit la source d'énergie la plus rentable.

On essaie donc de faire faire des choix au joueur, alors qu'il a conscience seulement du score global. A la fin, le nombre de mort et l'émission de CO2 de la partie sont divulguées au joueur. Cela incite le joueur à essayer de rejouer en prenant en compte ces aspects.

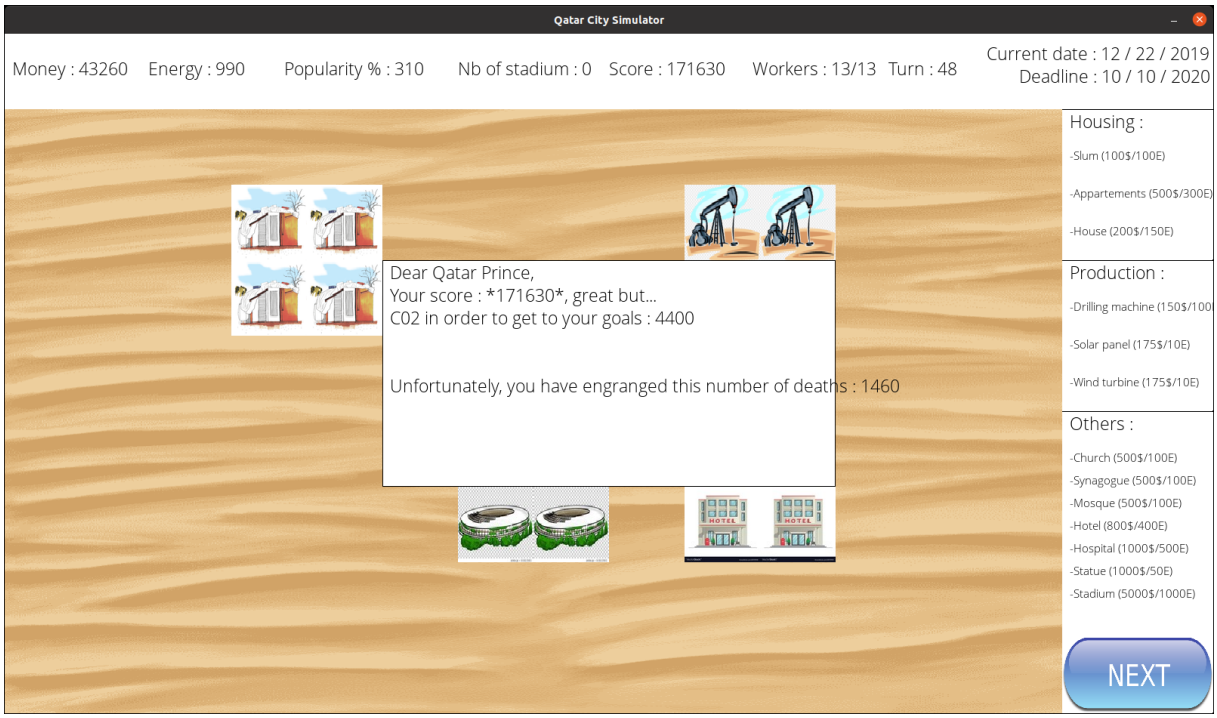
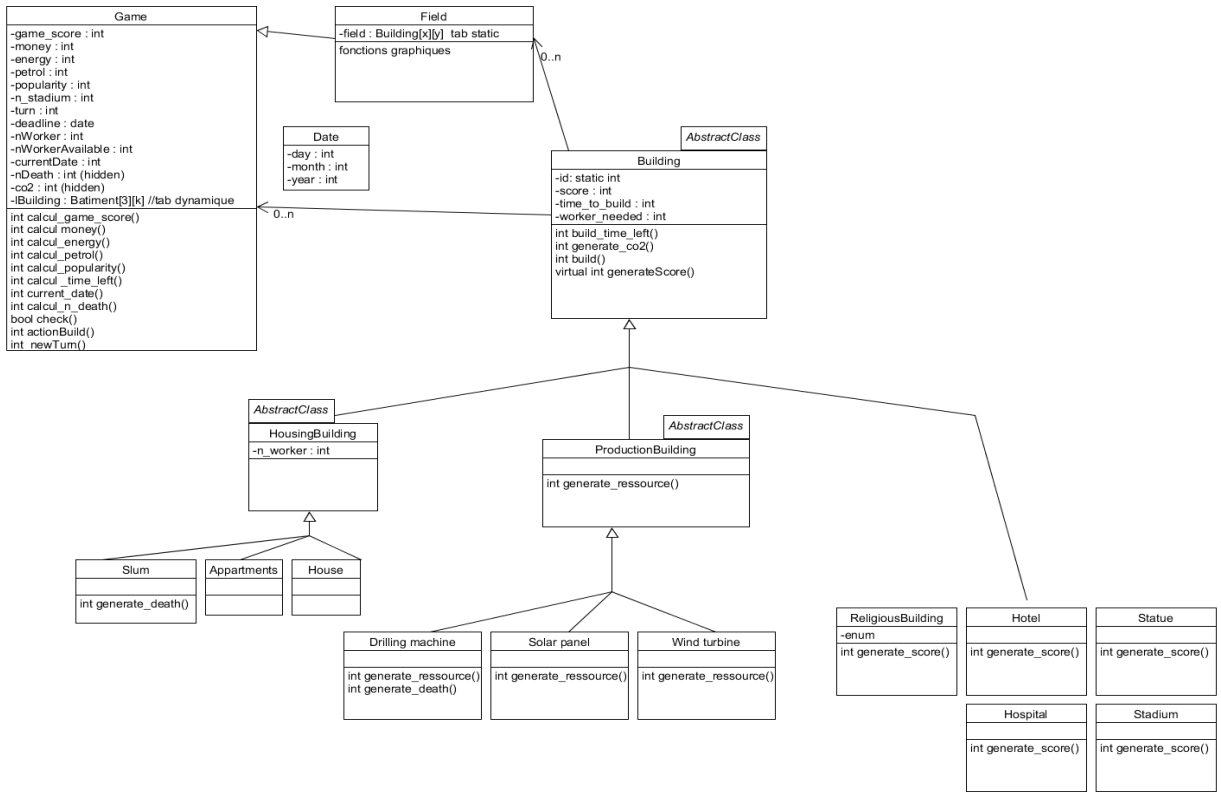


Figure 1: Texte de fin de partie gagnée

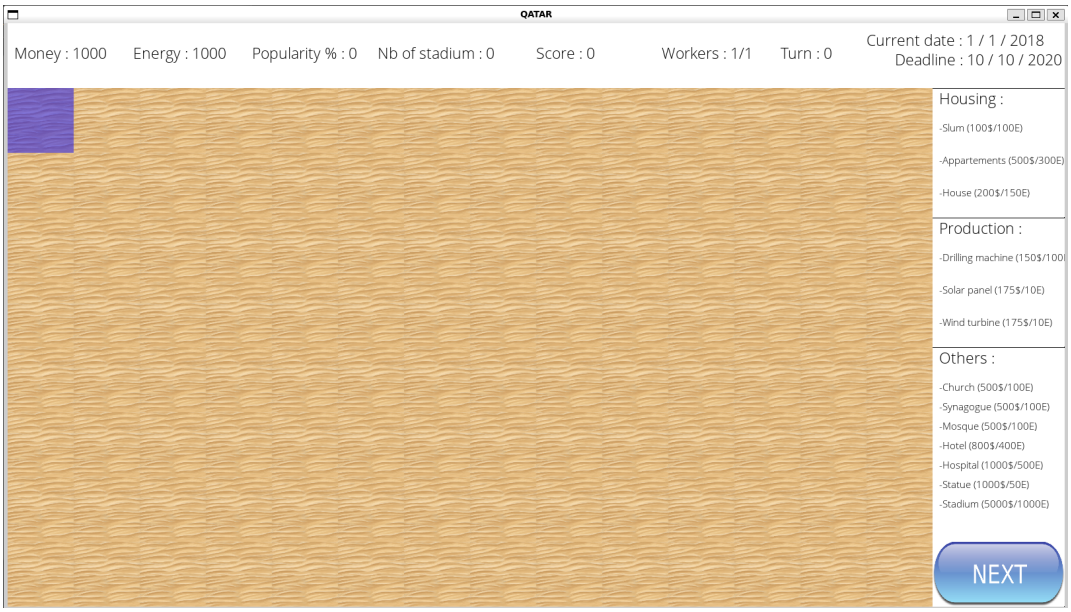
Il pourra essayer d'utiliser seulement des panneaux solaires ou des éoliennes pour avoir de l'énergie, et les hôpitaux permettent pour chaque hôpital construit de diviser le nombre de morts par deux. Il peut aussi juste choisir de ne pas loger ses travailleurs dans des bidonvilles.

<u>Voici le diagramme UML de correspondant :</u>



Comment se passe une partie :

Voici l'écran de jeu. On peut choisir une case, et cliquer sur le menu à droite sur ce que l'on veut construire.



Les ressources du joueur sont affichées en haut. On peut voir qu'on commence avec 1000 argent, 1000 énergie et un travailleur (« worker »).

En haut à droite il y a marqué la deadline et la date actuelle.
Le bouton permettant de changer de tour est en bas à gauche (« NEXT »).

Les outils pour y arriver :

Ce jeu a été fait en C++. On pense que la programmation orientée objet et très adaptée pour faire un jeu vidéo. C'était assez intuitif de coder ces concepts. A part des problèmes techniques, la conception s'est faite de manière fluide.

Les outils que nous avons trouvés particulièrement utiles sont :

- Le système de classes : une approche simple à utiliser pour faire interagir des éléments entre eux et organiser les données importantes.
- Les fonctions virtual : elles permettent de rendre le code plus lisible, beaucoup moins lourd à écrire et à lire. Appeler la même fonction car elle doit faire la même chose mais sur un objet différent est un gain de temps et de clarté.
- Les surcharges d'opérateur : pareil, c'est beaucoup plus intuitif de rajouter des jours à une date en utilisant « + », qu'appeler une fonction (par exemple).
- Les librairies : ça nous a permis de faire facilement une interface graphique simple.

Les parties du code qui nous plaisent :

Comment construire un bâtiment :

La construction d'un bâtiment se fait en plusieurs fonction.

La fonction Game::build() construit un bâtiment en fonction du paramètre rentré (une chaîne de caractères). Pour cela il appelle le constructeur par défaut des classes correspondantes au bâtiment demandé, et le stock dans les vecteurs de la classe Game où sont rangés tous les bâtiments construits (ou en construction).

```
int Game::checkMoney(int n){
    if(money >= n){
        return 1;
    } else {
        std::cout << "Not enough money" << std::endl;
        return 0;
    }
}

int Game::checkWorkers(int n){
    if(nWorkersAvailable >= n){
        return 1;
    } else {
        std::cout << "Not enough workers available" << std::endl;
        return 0;
    }
}

int Game::checkEnergy(int n){
    if(energy >= n){
        return 1;
    } else {
        std::cout << "Not enough energy" << std::endl;
        return 0;
    }
}
```

Mais pour appeler la fonction build qui va construire peut importe les ressources, il faut d'abord vérifier si on a tout ce qui est requis.

C'est à ça que servent les fonctions checkNWorkers(), checkEnergy() et checkMoney.

Ces fonctions prennent en paramètre la quantité qu'elles doivent comparer avec les ressource du joueur.

Puis finalement, on a la fonction qui rassemble les deux d'avants. Si toutes les fonctions check sont validées, alors on appelle build. C'est la fonction ActionBuild() qui prend en paramètre son coût en argent et en énergie et le nombre de travailleurs nécessaire.
(Même si on a finalement utilisé qu'un seul travailleur par construction pour tous les bâtiments).

```
int Game::ActionBuild(std::string typeBuilding, int w, int m, int e){
    if(checkWorkers(w)&&checkMoney(m)&&checkEnergy(e)){
        if(build(typeBuilding)){
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}
```

De la commande à la partie graphique

Pour passer d'un jeu se déroulant dans la commande comme le test 7 de main.cpp, à une partie graphique, on n'a presque pas eu de problèmes.

Voici ce que nous a un peu bloqué :

- Le jeu était beaucoup trop lent, on pouvait à peine cliquer. C'est parce que les images qu'on avait utilisé en texture étaient trop grandes. Après les avoir réduit en taille ça a fonctionné normalement.
- L'affichage final. C'était difficile d'afficher en partie graphique le message de fin pour plusieurs raisons. D'abord fin() ne renvoyait rien à la base. On a décidé de lui faire renvoyer une chaîne de caractères en plus de l'afficher dans le terminal.

  Pour les retours à la ligne on s'est rendu compte que pour l'affichage graphique, \n ne marchait pas. C'est pourquoi on a décidé de couper la chaîne de caractères retournée et donc d'afficher ligne par ligne. En effet à chaque re affichage on retourne à la ligne.

  Pour cela on a utilisé les fonctions std::string::substr() et find() pour trouver l'index d'un caractère ou d'une sous-chaîne.

  Mais il y a des phrases à ne pas forcément mettre à la fin.. comme par exemple la phrase qui indique que peu de CO2 a été émis. Donc on a utilisé find() qui renvoie -1 si la fonction ne trouve pas la chaîne demandée.

  Voici un bout de ce code :

```
font = TTF_OpenFont("texture/sans.ttf", 23);
std::string original = game->fin().c_str();
std::string sub1 = original.substr(0,18);
int pos1 = original.find("but...");
std::string sub2 = original.substr(19,pos1-13);
int pos2 = original.find("YNY");
```

YNY était utilisé pour nous repérer dans la chaîne par exemple.

Comment installer les librairies et exécuter le code :

A partir d’une invite de commande sous Linux dans le fichier de notre jeu :

Rentrez ces lignes pour installer la librairie STDL2 :

sudo apt-get install libsdl2-dev

sudo apt-get install libsdl2-image-dev

sudo apt-get install libsdl2-ttf-dev

Puis pour compiler et créer l’exécutable rentrez simplement :

make

Et pour jouer :

./out

Si vous voulez voir comment on a fait un test, on vous conseille de regarder le test 7 qui est une simulation de ce qu’un joueur aurait pu faire.
Pour tester il faut mettre en commentaire le reste et laisser que le test 7 et return 0 dans main.cpp.