```
#include "Model.h"
#include "Ninja.h"
#include <iostream>
#include <fstream>
#include <cstdlib>

/*
 * ----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * Antoine BOULANGÃM-^I & Pierre_Elliot CABRERA wrote this file. As long as you
retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return
 * ----------------------------------------------------------------------------
 */


using namespace std;

//=======================================
// Constructeurs
//=======================================
Model::Model(int w, int h)
    : _w{w}, _h{h}
    , _hp{NUMBER_HP}
    , _damaged{false}
    , _score{0}
    , _money{0}
    , _invincibility{false}
    , _difficulty{EASY}
    , _introduction{true}
    , _menu{false}
    , _mort{false}
    , _highscore{false}
    , _menuHighscore{false}
{
    srand(time(NULL));
    int bx, by, bw, bh;
    bx = 50;
    bh = 75;
    bw = 54;
    by = GROUND_HEIGHT - bh;
    _ninja = new Ninja(bx,by,bw,bh,0,0);
    _clockObstacle.restart();
    _clockBonus.restart();
    _repopTimeObstacle = sf::seconds(rand() % (MAXIMUM_SPAWN_OBSTACLE - MINIMUM_
SPAWN_OBSTACLE) + MINIMUM_SPAWN_OBSTACLE);
    _repopTimeBonus = sf::seconds(rand() % (MAXIMUM_SPAWN_BONUS - MINIMUM_SPAWN_
BONUS) + MINIMUM_SPAWN_BONUS);
}

//=======================================
// Destructeurs
//=======================================
Model::~Model(){
    delete _ninja;
    for(auto it : _elements){
        delete it;
    }
    for(auto it : _elementsDeleted){
        delete it;
    }
}

//=======================================
// Calcul la prochaine Ã©tape
//=======================================
void Model::nextStep(){
    if (_hp == 0)
```



```
            _mort = true;
    else if(!_menu){
        // Prochaine Ã©tape dans le cas oÃ¹ l'on est dans le jeu

        // Met Ã  jour les modifications du controller de la View lorsque le jou
eur n'est plus en Ã©tat 'SAUT'
        if(getNinjaStatus() != JUMPING){
            if(getNinjaToStopX()){
                stopNinja(true);
                setNinjaToStopX(false);
            }
            if(getNinjaToStopY()){
                stopNinja(false);
                setNinjaToStopY(false);
            }
        }

        // Supprime les objets du tableau d'obstacles et de bonus lorsque ceux-c
i sortent de l'Ã©cran
        auto it = _elements.begin();
        while(it != _elements.end()){
            (*it)->move();
            if((*it)->getX() + (*it)->getW() < 0){
                delete (*it);
                _elements.erase(it);
            }else
                ++it;
        }

        // GÃ¨re le mouvement du joueur et la vÃ©rification de collision
        _ninja->move();
        collisionNinja();

        // DÃ©termine s'il est temps de crÃ©er un nouvel obstacle
        // Si c'est le cas, dÃ©termine le temps Ã  atteindre jusqu'au prochain o
bstacle
        if (_clockObstacle.getElapsedTime() > _repopTimeObstacle)
        {
            addElement(true);
            do{
                _repopTimeObstacle = sf::seconds((1.0) * (rand() % ((MAXIMUM_SPA
WN_OBSTACLE - MINIMUM_SPAWN_OBSTACLE) * 100) + MINIMUM_SPAWN_OBSTACLE) / 100);
            }while(_repopTimeObstacle < sf::seconds(0.30 * (HARD - _difficulty))
);
            _clockObstacle.restart();
        }

        // DÃ©termine s'il est temps de crÃ©er un nouvel bonus
        // Si c'est le cas, dÃ©termine le temps Ã  atteindre jusqu'au prochain b
onus
        if (_clockBonus.getElapsedTime() > _repopTimeBonus)
        {
            addElement(false);
            _repopTimeBonus = sf::seconds(rand() % (MAXIMUM_SPAWN_BONUS - MINIMU
M_SPAWN_BONUS) + MINIMUM_SPAWN_BONUS);
            _clockBonus.restart();
        }

        // DÃ©termine si le temps d'activitÃ© du bonus 'VOL' est arrivÃ© Ã  son
terme
        if(_ninja->getStatus() == SOARING){
            if(_clockFly.getElapsedTime() > LAPSE_FLY){
                _ninja->setStatus(TRANSITIONNING);
                moveNinjaY(false);
            }
        }

        // DÃ©termine si le temps d'activitÃ© du bonus 'INVINCIBILITÃM-^I' est a
rrivÃ© Ã  son terme
```

```
        if(_clockInvincibility.getElapsedTime() > LAPSE_INVINCIBILITY){
            _invincibility = false;
        }

        // IncrÃ©mente le score
        addScore(1);
    }
}

//=====================================
// Definit la valeur du mouvement
// en ordonnÃ©es de la ninja
//=====================================
void Model::moveNinjaY(bool up) {
    if (up)
        _ninja->setMvtY(10);
    else
        _ninja->setMvtY(-10);
}

//=====================================
// ArrÃªte la ninja (mouvement d'abscisse)
//=====================================
void Model::stopNinja(bool abscissa){
    if(abscissa)
        _ninja->setMvtX(0);
    else
        _ninja->setMvtY(0);
}

//=====================================
// Ajoute un MovableElement
// (Bonus ou Obstacle)
//=====================================
void Model::addElement(bool obstacle){
    if(obstacle){
        TYPES elemType;
        int elemX, elemY, elemW, elemH, elemMvtX;

        if(_ninja->getStatus() == SOARING){
            // Si le joueur est en possession du bonus de type 'VOL', les obstac
les gÃ©nÃ©rÃ©s sont tous de type 'VOLANT' et ont une taille augmentÃ©e

            elemW = 50;
            elemH = 50;
            elemY = rand() % (GROUND_HEIGHT - elemH);
            elemType = FLYING;
        }else{
            // CrÃ©e un obstacle de type alÃ©atoire parmi les trois disponibles:
 'PETIT', 'GRAND' ou 'VOLANT'

            int typeNumber = rand() % NUMBER_OBSTACLE;
            switch(typeNumber){
                case 0 :
                    elemW = 25;
                    elemH = 50;
                    elemY = GROUND_HEIGHT - elemH;
                    elemType = SMALL;
                    break;
                case 1:
                    elemW = 50;
                    elemH = 50;
                    elemY = GROUND_HEIGHT - elemH;
                    elemType = BIG;
                    break;
                case 2:
                    elemW = 25;
                    elemH = 25;
                    elemY = rand() % (GROUND_HEIGHT - elemH - 271) + 271;
```



```
                    elemType = FLYING;
                    break;
            }
        }

        elemX = 1200;
        elemMvtX = ((-1) * (rand() % 5 + 2)) - _difficulty;

        // DÃ©termine si la place occupÃ©e par l'obstacle est en collision avec
un obstacle ou bonus dÃ©jÃ  existant
        bool freeSpace = true;
        sf::IntRect newElement(elemX, elemY, elemW, elemH);
        for(auto elem : _elements){
            sf::IntRect element(elem->getX(), elem->getY(), elem->getW(), elem->
getH());
            if(newElement.intersects(element))
                freeSpace = false;
        }

        // Il n'y a crÃ©ation de l'obstacle que s'il n'existe aucun conflit de s
uperposition
        if(freeSpace){
            AutonomousElement * elem = new AutonomousElement(elemX, elemY, elemW
, elemH, elemMvtX, 0, SCORE_MALUS, elemType);
            _elements.push_back(elem);
            _newElements.push_back(elem);
        }

    }else{
        // CrÃ©e un obstacle de type alÃ©atoire parmi les cinq disponibles: 'SOI
N', 'SCORE', 'VOL', 'INVINCIBILITÃ¨^I', 'COIN'
        // a une place disponible

        int typeNumber = rand() % NUMBER_BONUS;

        int elemX, elemY, elemW, elemH, elemMvtX;
        bool freeSpace;
        TYPES typeName;

        elemW = 25;
        elemH = 25;
        elemX = 1200;
        elemMvtX = -5 - _difficulty;


        do{
            freeSpace = true;
            elemY = rand() % (GROUND_HEIGHT - elemH - 271) + 271;

            sf::IntRect newElement(elemX, elemY, elemW, elemH);
            for(auto elem : _elements){
                sf::IntRect element(elem->getX(), elem->getY(), elem->getW(), el
em->getH());
                if(newElement.intersects(element))
                    freeSpace = false;
            }
        }while(!freeSpace);

        switch(typeNumber){
            case 0 :
                typeName = HEAL;
                break;
            case 1:
                typeName = SCORE;
                break;
            case 2:
                typeName = FLY;
                break;
            case 3:
```

```
                typeName = INVINCIBILITY;
                break;
            case 4:
                typeName = COIN;
        }
        AutonomousElement * elem = new AutonomousElement(elemX, elemY, elemW, el
emH, elemMvtX, 0, SCORE_BONUS, typeName);
        _elements.push_back(elem);
        _newElements.push_back(elem);
    }

}

//=====================================
// Vide le tableau de nouveaux Ã©lements
//=====================================
void Model::clearNewElements(){
    _newElements.clear();
}

//=====================================
// Vide le tableau de nouveaux Ã©lements Ã  dÃ©truire
//=====================================
void Model::clearNewElementsDeleted(){
    _newElementsDeleted.clear();
}

//=====================================
// Augmente la vitesse des obstacles et bonus
//=====================================
void Model::addSpeed(){
    for(auto it : _elements){
        it->setMvtX(it->getMvtX() * 1.5);
    }
}

//=====================================
// Determine s'il y a une collision
// entre la ninja et un bonus ou
// obstacle et applique les effets correspondants
//=====================================
void Model::collisionNinja(){
    auto it = _elements.begin();
    while(it != _elements.end())
    {
        if(_ninja->collision(*it)){

            int score = (*it)->getScore();
            switch((*it)->getType()){
                case BIG:
                case SMALL:
                case FLYING:
                    if(!_invincibility){
                        addScore(score);
                        changeLife(HP_LOSS);
                    }
                    break;
                case HEAL:
                    addScore(score);
                    changeLife(HP_HEALED);
                    break;
                case SCORE:
                    addScore(score + SCORE_BONUS_SCORE);
                    break;
                case FLY:
                    addScore(score);
                    setNinjaToStopX(true);
                    setNinjaToStopY(true);
                    setNinjaStatus(SOARING);
```



```
                    _clockFly.restart();
                    break;
                case INVINCIBILITY:
                    _invincibility = true;
                    addScore(score);
                    _clockInvincibility.restart();
                    break;
                case COIN:
                    _money++;
                    addScore(score);
                    break;
            }
            _elementsDeleted.push_back(*it);
            _newElementsDeleted.push_back(*it);
            _elements.erase(it);
        }
        else
            ++it;
    }
}

//=====================================
// Definit la valeur du mouvement
// en abscisse de la ninja suivant
// sa direction
//=====================================
void Model::moveNinjaX(bool forward) {
    if (forward)
        _ninja->setMvtX(10);
    else
        _ninja->setMvtX(-10);
}

//=====================================
// Ajoute une certaine valeure au score
//=====================================
void Model::addScore(int value){
    _score += value;
}

//=====================================
// Ajoute une certaine valeur au total
// de points de vie, valeur pouvant
// Ãªtre nÃ©gative
//=====================================
void Model::changeLife(int gain){
    _hp += gain;

    if(_hp > NUMBER_HP)
        _hp = NUMBER_HP;

    if(gain < 0)
        _damaged = true;
}

//=====================================
// Mutateurs
//=====================================
void Model::setNinjaDirection(bool forward){ _ninja->setDirection(forward); }

void Model::setNinjaToStopX(bool toStop){ _ninja->setToStopX(toStop); }

void Model::setNinjaToStopY(bool toStop){ _ninja->setToStopY(toStop); }

void Model::setNinjaXBeginJmp(int value){ _ninja->setXBeginJmp(value); }

void Model::setNinjaYBeginJmp(int value){ _ninja->setYBeginJmp(value); }

void Model::setNinjaToChangeDirection(NINJA_CHANGE_DIRECTION newDirection){ _nin
```

```
ja->setToChangeDirection(newDirection); }

void Model::setNinjaReduction(bool toReduce){ _ninja->setReduction(toReduce); }

void Model::setNinjaStatus(NINJA_STATUS value){ _ninja->setStatus(value); }

void Model::setMenu(bool menu){ _menu = menu; }

void Model::setDead(bool mort){ _mort = mort; }

void Model::setIntroduction(bool introduction){ _introduction = introduction; }

void Model::setHighscore(bool highscore){ _highscore = highscore; }

void Model::setMoney(int money){ _money = money; }

void Model::setMenuHighScore(bool menuHighscore){ _menuHighscore = menuHighscore
; }

//=======================================
// Accesseurs
//=======================================
bool Model::getNinjaToStopX() const{ return _ninja->getToStopX(); }

bool Model::getNinjaToStopY() const{ return _ninja->getToStopY(); }

int Model::getHp(){ return _hp; }

int Model::getScore(){ return _score; }

int Model::getNinjaToChangeDirection() const{ return _ninja->getToChangeDirectio
n(); }

void Model::getNinjaPosition(int &x, int &y) const {
    x = _ninja->getX();
    y = _ninja->getY();
}

void Model::getNinjaDimension(int &w, int &h) const {
    w = _ninja->getW();
    h = _ninja->getH();
}

void Model::getNinjaSpeedX(int &mvtX) const { mvtX = _ninja->getMvtX(); }

void Model::getNinjaSpeedY(int &mvtY) const { mvtY = _ninja->getMvtY(); }

NINJA_STATUS Model::getNinjaStatus() const{ return _ninja->getStatus(); }

std::vector<const AutonomousElement *> Model::getNewAutonomousElements() const{
return _newElements; }

std::vector<AutonomousElement *> Model::getAutonomousElements() const{ return _e
lements; }

std::vector<const AutonomousElement *> Model::getNewElementsDeleted() const{ ret
urn _newElementsDeleted; }

std::vector<AutonomousElement *> Model::getElementsDeleted() const{ return _elem
entsDeleted; }

void Model::changeDamaged(){ _damaged = !_damaged; }

bool Model::getDamaged(){ return _damaged; }

bool Model::getMenu() const { return _menu; }

bool Model::getDead() const { return _mort; }
```



```
bool Model::getInvincibility() const { return _invincibility; }

int Model::getMoney() const { return _money; }

bool Model::getHighscore() const { return _highscore; }

bool Model::getIntroduction() const { return _introduction; }

bool Model::getMenuHighscore() const { return _menuHighscore; }
```