Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Алтайский государственный технический университет им. И.И. Ползунова»

Факультет информационных технологий Кафедра прикладной математики

Отчет защищен с оценкой

Руководитель от вуза А.Ю. Андреева

(подпись) (и.о., фамилия)

« » 2022 г.

(дата)

## Отчет об учебной (технологической) практике

«Симуляция поиска и сбора ресурсов»

УП 09.03.04.1.2 О

|  |  |  |
| --- | --- | --- |
| Студент группы | Немчинов Виктор Ильич | ПИ-02 |
| Руководитель практики | доцент, к.ф.-м.н. | и.о., фамилия  А.Ю. Андреева |
|  | должность, ученое звание | и.о., фамилия |

#### Барнаул 2022

ФГБОУ ВО «Алтайский государственный технический университет им. И. И. Ползунова»

Кафедра «Прикладная математика»

###### Индивидуальное задание

###### На учебную (технологическую (проектно-технологическую) практику)

(вид и тип практики по УП)

## студенту Немчинов Виктор Ильич группы ПИ-02

(Ф.И.О.)

###### График проведения практики

|  |  |  |
| --- | --- | --- |
| **№**  **п/п** | **Наименование работ, выполняемых на практике** | **Сроки выполнения** |
| 1 | Разработка техническое задание |  |
| 2 | Проектирование объектной модели |  |
| 3 | Реализация модели в виде программы с графическим  интерфейсом |  |
| 4 | Написание отчета и защита практики |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Руководитель практики от университета | | (подпись) | АндрееваА.Ю., доцент каф. ПМ  (Ф.И.О., должность) |
| Задание принял к исполнению | (подпись) | | (Ф.И.О.) |

###### Инструктаж по ОТ, ТБ, ПБ, ПВТР

Инструктаж обучающегося по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, а также правилами внутреннего трудового распорядка проведен «20» июня 2022 г.

|  |  |  |
| --- | --- | --- |
| Руководитель практики от  Университета |  | Андреева А.Ю. доцент каф. ПМ |
|  | (подпись) | (Ф.И.О., должность) |

**Аннотация**

Отчет об учебной практике содержит описание программы, реализующей моделирование поиска и сбора ресурсов: техническое задание, структуру данных, структуру файлов, описание программного продукта. Код программы на языке C++ CLI размещен в репозитории на Github и в приложении Б. В приложении А приведены снимки экранных форм программы.

Отчет содержит 40 страниц, 3 рисунка, 4 источника литературы.

**Оглавление**

[Введение 5](#_bookmark0)

1. [Техническое задание 6](#_bookmark1)
   1. [Терминология 6](#_bookmark2)
   2. [Описание процесса функционирования модели 7](#_bookmark3)
   3. [Требования к функциональности программы 7](#_bookmark4)
2. [Проект программного продукта 9](#_bookmark5)
   1. [Математическая модель 9](#_bookmark6)
   2. [Диаграмма классов 11](#_bookmark7)
   3. [Жизненный цикл объектов модели 12](#_bookmark8)
3. [Описание программного продукта 12](#_bookmark9)
   1. [Выбор средств реализации 12](#_bookmark10)
   2. [Описание классов 13](#_bookmark11)

[Заключение 16](#_bookmark12)

[Список использованных источников 17](#_bookmark13)

[Приложение A. Снимки экранных форм пользовательского интерфейса 18](#_bookmark14)

[Приложение Б. Исходный код 20](#_bookmark15)

# Введение

Моделирование – основной метод исследований объектов, процессов или явлений, с целью получения объяснений этих явлений, а также для предсказания развития явлений и процессов, интересующих исследователей. Задача моделирования - выявить главные, характерные черты явления или процесса, его особенности, его поведение. Моделирование применяется в разных областях человеческой жизни: в медицине, демографии, страховании, социологии, в научно-исследовательской деятельности, в повседневной жизни и даже в компьютерных играх.

Модель (объекта – оригинала) – вспомогательный объект, отражающий наиболее существенные для исследования закономерности, суть, свойства, особенности строения и функционирования объекта-оригинала. Задачи моделирования:

* Понять сущность изучаемого объекта
* Научиться управлять объектом и определять наилучшие способы управления
* Решать прикладные задачи

Также моделирование позволяет прогнозировать прямые или косвенные последствия того или иного процесса или явления. Это одна из самых главных задач моделирования в случае с распространением вирусов.

# Техническое задание

# Терминология

**Сборщик ресурсов** - объект, цель которого найти ресурс и доставить его в хранилище, соответствующее типу найденного ресурса.

**Собираемый предмет –** объект, который в ходе выполнения программы будет собран. Является виртуальным. Базовый для ресурса пищи и руды.

**Ресурс пищи** – объект, который обладает определенным запасом единиц определенного типа, которые в последствии будут собраны сборщиком и доставлены в определенное хранилище.

**Ресурс руды** – объект, аналогичный ресурсу пищи, но запас, которого выражается в иных единицах.

**Хранилище** – объект, обладающий большим объемом, суть которого в хранении единиц определенного ресурса, которые доставляются сборщиками. При заполнении хранилища его показатель объема хранимых единиц сбрасывается. Существует два типа хранилища: одно предназначено для пищи и располагается в верхнем левом углу карты, другое для руды в нижнем правом.

**Препятствие** – объект, располагаемый на карте, который избегают сборщики ресурсов в процессе поиска, но игнорирующие его при доставке в хранилище.

**Граница карты** – четыре объекта, отвечающие за соответствующие стороны карты, ограничивающие движение сборщиков ресурсов.

**Поле сбора** – невидимое поле, которое находится перед сборщиком ресурсов и сонаправлено его вектору движения. Служит для определения типа объекта, с которым соприкасается это поле.

**Поле поиска** - представляет собой невидимый длинный луч, который появляется перед сборщиком ресурсов после заполнения его рюкзака определенного ресурса. Служит для определения маршрута до хранилища.

**Рюкзак сборщика** – в распоряжении сборщика находятся два рюкзака малого объема для сбора ресурсов. Один рюкзак для пищи, другой для руды. Показателем их заполненности являются две полоски разного цвета на сборщике.

# Описание процесса функционирования модели

Модель функционирует в виде игры, на главной странице нам представлена карта, на которой уже изначально размещены два хранилища: в левом верхнем углу находится хранилище для пищи, в нижнем левом для руды.

Для начала симуляции необходимо создать сборщиков, ресурсы для сбора и при необходимости препятствия. Для каждого объекта предусмотрены соответствующие клавиши: сборщик – пробел, ресурс пищи – 1, ресурс руды – 2, препятствие – 3.

Так же в программе предусмотрен сброс добавленных объектов и обнуление хранилищ. Для этого необходимо нажать клавишу 4.

Сборщики начинают доставку сразу после заполнения одного из рюкзаков, но также предусмотрена возможность начала доставки по команде. Для этого необходимо нажать клавишу 5. В этом случае все сборщики при наличии любого ресурса начнут движение к хранилищу.

При заполнении хранилища его объем сбрасывается и начинается заполнение по новой, что делают симуляцию бесконечной.

# Требования к функциональности программы

В программе имеется графический интерфейс: одна главная форма, на которой расположена карта, на которой изначально находятся лишь пустые хранилища и границы карты.

На поле может быть отображено до 5 типов объектов, при этом по мере выполнения программы они осуществляют взаимодействие друг с другом. При запуске на поле располагаются лишь объекты, ограничивающие карту и хранилища для пищи и руды. Остальные объекты создаются при нажатии соответствующих клавиш.

Предусмотрены следующие типы объектов:

* Граница карты (border. Четыре типа, каждый из которых соответствует определенной стороне карты);
* Хранилище (storage);
* Сборщик (human);
* Собираемый предмет (collectible item);

1. Ресурс пищи (food);
2. Ресурс руды (ore).

* Препятствие (obstacle);

Симуляция начинается с создания сборщиков и ресурсов, а также при необходимости препятствий.

Далее, сборщики начинают движение по карте, отскакивая от стен и препятствий в соответствии с правилом отражения. Шаг производится по тику таймера. При каждом шаге перед сборщиком создается поле небольшого размера, служащее для определения объекта перед сборщиком. Если в поле находится объект, то в главную форму, в которой находятся все объекты поступает запрос от сборщика на определение встреченного объекта, после чего главная форма сообщает сборщику о действиях, которые необходимо совершить, в соответствии с найденным объектом.

При встрече объектов типа «граница карты» или «препятствие» происходит отскок в соответствии с законом «угол падения углу отражения». При соприкосновении с хранилищами в режиме поиска, сборщики отскакивают в случайном направлении в противоположную сторону.

При встрече объектов типа «Собираемый предмет», происходит определение типа собираемого предмета, и заполнение соответствующего рюкзака сборщика. Различается два типа собираемых предметов: ресурс пищи и ресурс руды. Заполненность рюкзака отображается полоской на сборщике: синяя полоска отвечает за заполненность рюкзака руды, зеленая за рюкзак пищи. При заполнении какого-либо из рюкзаков мгновенно начинается движение к хранилищу, тип которого соответствует только что заполненному рюкзаку.

Также при сборе ресурса графический объект, отвечающий за ресурс, уменьшается при уменьшении у него количества собираемых единиц.

Направление до хранилища определяется путем кастования луча вокруг своей оси. Кастование начинается с 0 градусов. При отсутствии коллизии луча с хранилищем происходит поворот сборщика на 1 градус по часовой стрелке, после чего процедура повторяется до определения угла до хранилища. Программой не предусмотрен случай, при котором хранилище не будет найдено.

Движение к хранилищу может начаться не только по заполненности рюкзака, но также и по прямой команде, осуществляемой путем нажатия клавиши 5. При этом сборщик начнет двигаться сначала к хранилищу пищи, естественно при наличии у него пищи, и только после повторного нажатия к хранилищу руды. При отсутствии у сборщика пищи - движение начнется к хранилищу с рудой. При отсутствии у сборщика ресурсов – сборщик проигнорирует команду. Также сборщик игнорирует команду, если уже находится в процессе доставки.

При движении до хранилища сборщик полностью игнорирует любые встреченные объекты на своем пути, таким образом двигаясь по прямой.

При достижении хранилища, сборщик передает все содержимое соответствующего рюкзака в хранилище. После передачи ресурсов, полоска определенного рюкзака обнуляется, а полоска, отвечающая за заполненность хранилища – увеличивается. При заполнении хранилища, полоска хранилища обнуляется.

Также программой предусмотрено ускорение и замедление движения сборщиков, путем уменьшения интервала таймера. Ускорение осуществляется клавишей 6, замедление клавишей 7.

Объекты типа «Сборщик», «Собираемый предмет» и «Препятствие» являются динамическими и могут быть передвинуты мышью.

# Проект программного продукта

# Математическая модель

Карта на которой происходит симуляция – *QGraphicsView* и *QGraphicsScene*. *QGraphicsView* – поле на окне, предназначенное для отображения контента, располагаемого на *QGraphicsScene*.

##### Все графические объекты сцены являются наследниками класса *QGraphicsItem* и *QObject*.

##### Классы *CollectibleItem* и Storage являются виртуальными. От класса *CollectibleItem* наследуются классы *Food* и *Ore*, в них лишь переопределены методы отрисовки. От класса *Storage* наследуются классы *FoodStorage* и *OreStorage*, в них так же переопределены только методы отрисовки.

##### Взаимодействие объектов

Сборщики перемещаются хаотично по карте. Их движение изменяется только при соприкосновении с препятствиями, хранилищами или границами карты. Их перемещение осуществляется командой *advance()*, которая срабатывает при каждом тике таймера главного окна. Главное окно оповещает сборщика о каждом тике с помощью сигнал-слота *connect(timer, SIGNAL(timeout()), scene, SLOT(advance())).* После каждого шага сборщик строит перед собой поле с помощью полигонов, после чего сигнал-слотом *connect(human, &Human::signalCheckItem, this, &MainWindow:: slotCheckFacedObject)* запрашивает главное окно необходимых действиях с найденным объектом, попавшим в поле. В слоте *slotCheckFacedObject* главного окна осуществляется проверка всех объектов на сцене на совпадение с указателем на найденный объект, после чего в этом же слоте выполняется определенное действие со сборщиком и найденным им предметом. К примеру, при определении найденного объекта как «Препятствие» осуществляется метод сборщика *DoCollision()*. Таким образом, слот *slotCheckFacedObject* является методом, в котором определяются все взаимодействия между сборщиком или любым встреченным им объектом.

##### Сущности

**Сборщик ресурсов (human)**

Характеристики сборщика:

* startX - начальная позиция сборщика;
* startY – начальная позиция сборщика;
* angle – азимут движения сборщика;
* speed – скорость сборщика;
* location – позиция на карте;
* itemIsCollidingOnSpawn – проверка при появлении на коллизию;
* isMovingToStorage – движется ли сборщика к хранилищу;
* isMovingToStorageNotFull – движется ли к хранилищу по команде;
* localFoodStorage – ссылка на хранилище пищи, для определения направления;
* localOreStorage - ссылка на хранилище руды, для определения направления.
* packageOfFood – рюкзак с пищей (до 100 единиц);
* packageOfOre – рюкзак с рудой (до 100 единиц);

**Собираемый предмет (collectible item)**

Характеристика предмета:

* startX - начальная позиция предмета;
* startY- начальная позиция предмета;
* pointsToCollect – очки оставшиеся для сбора;
* itemIsCollidingOnSpawn – проверка при появлении на коллизию;

**Препятствие (obstacle)**

Характеристика предмета:

* obstacleColor – цвет препятствия;
* itemIsCollidingOnSpawn – проверка при появлении на коллизию;
* width – ширина препятствия;
* height - высота;
* startX - начальная позиция предмета;
* startY - начальная позиция предмета;
* location - позиция препятствия;

**Хранилиoе (storage)**

Характеристика предмета:

* quantityOfResources – количество ресурсов в хранилище (до 1500);
* startX - начальная позиция предмета;
* startY - начальная позиция предмета;

##### Взаимодействие объектов

**Сборщик - Препятствие**

При попадании препятствия в поле сбора сборщика, происходит запрос в главную форму на определение встреченного объекта. После определения того, что это препятствие, сборщик из главной формы получает информацию о том, с какой стороны препятствия было соприкосновение, и в соответствии с этим происходит отражение вектора движения.

**Сборщик – Граница Карты**

При попадании границы карты в поле сборщика, происходит запрос в главную форму на определение встреченного объекта. После определения сборщику передается информация о том, какая из границ была встречена, после чего происходит соответствующее отражение вектора движения.

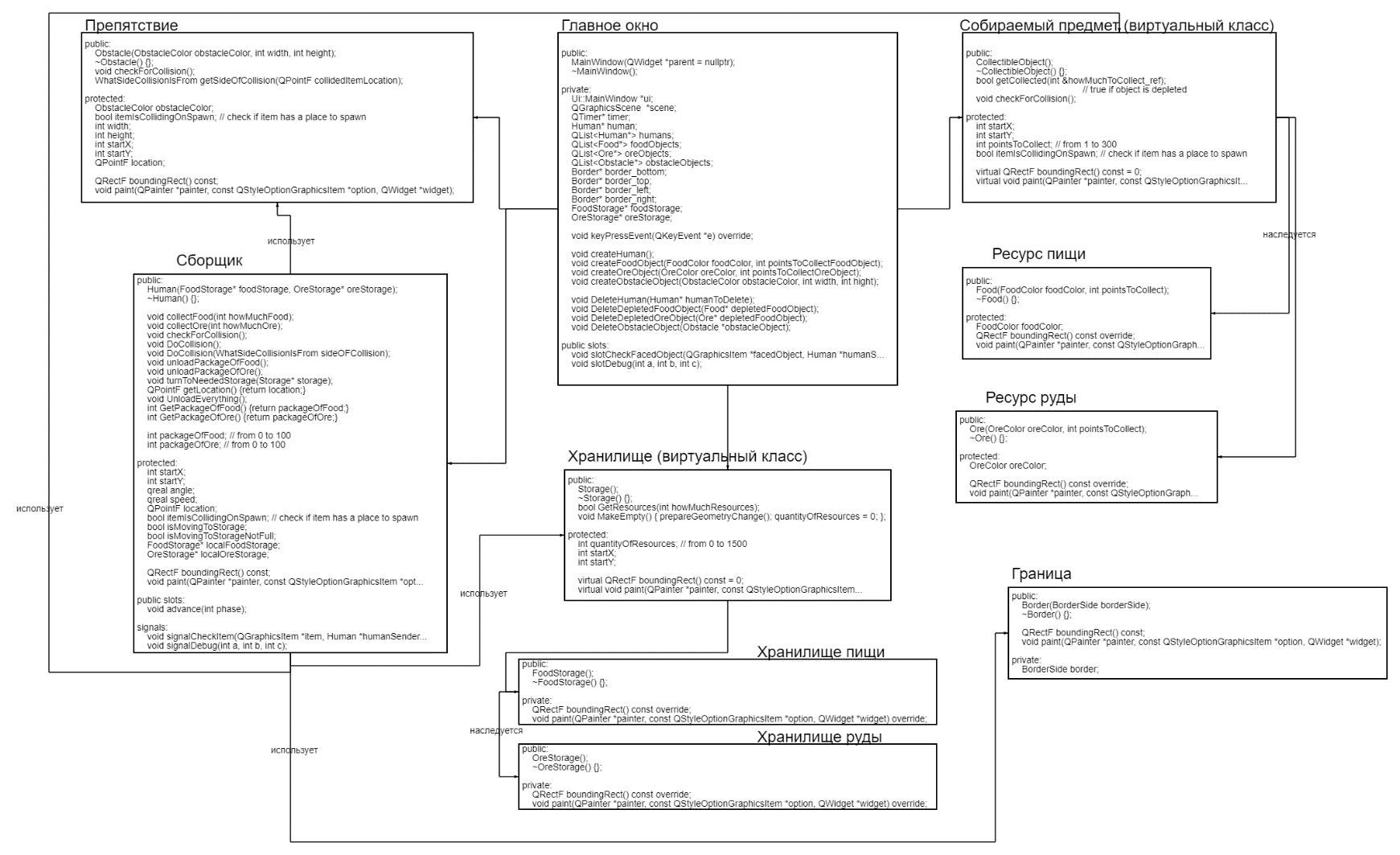
**Сборщик – Собираемый предмет**

При попадании собираемого предмета в поле сборщика, происходит запрос в главную форму на определение встреченного объекта. После определения типа встреченного объекта, происходит забор его ресурсов в соответствующий рюкзак. Если ресурс содержит единиц меньше, чем способен вместить рюкзак – сборщик собирает последнее и объект удаляется с карты. В обратном случае – собираемый предмет уменьшается в соответствии с количество оставшихся очков.

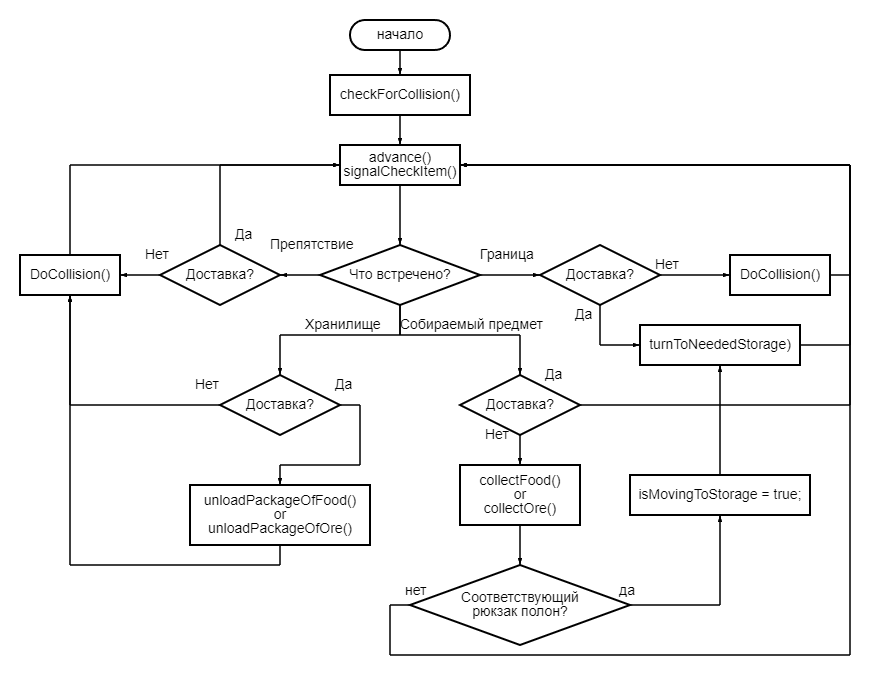
**Сборщик – Хранилище**

При попадании хранилища в поле сборщика, происходит запрос в главную форму на определение встреченного объекта. После определения типа встреченного объекта, если сборщик встречает хранилище при доставке ресурсов, то происходит передача всех имеющихся ресурсов определенного типа хранилищу. Если сборщик сталкивается с хранилищем в режиме поиска – происходит отскок в противоположном случайном направлении.

# Диаграмма классов



# Жизненный цикл объектов модели



*Жизненный цикл сборщика ресурсов*

# Описание программного продукта

# Выбор средств реализации

Данная модель была реализована на языке С++ с использованием среды разработки программного обеспечения Qt Creator. Язык С++ - компилируемый статически типизированный язык программирования общего назначения.

Программа соответствует всем парадигмам объектно-ориентированного программирования.

Использовались стандартные библиотеки языка Си:

**stdio.h**— стандартный заголовочный файл ввода-вывода, содержащий определения макросов, константы и объявления функций и типов, используемых для различных операций стандартного ввода и вывода

**stdlib.h** — заголовочный файл стандартной библиотеки языка С, который содержит в себе функции, занимающиеся выделением памяти, контролем процесса выполнения программы, преобразованием типов и другие.

**time.h** — заголовочный файл стандартной библиотеки языка программирования C, содержащий типы и функции для работы с датой и временем. В данном ПО использовался для генерации псевдослучайных чисел.

**string.h** — заголовочный файл стандартной библиотеки языка С, содержащий функции для работы со строками, оканчивающимися на 0, и различными функциями работы с памятью.

И так же использовались встроенные библиотеки программы Qt Creator.

# Описание классов

### Классы:

**MainWindow** – является основным классом данной симуляции, в котором содержатся главная сцена отображения объектов и все остальные объекты.

**Класс содержит:**

private:

QGraphicsScene \*scene – саму сцену;

QTimer\* timer – таймер, по которому осуществляется движение сборщиков;

Human\* human – указатель для создания сборщика;

QList<Human\*> humans – лист всех сборщиков;

QList<Food\*> foodObjects – лист всех ресурсов пищи;

QList<Ore\*> oreObjects – лист всех ресурсов руды;

QList<Obstacle\*> obstacleObjects – лист всех ресурсов препятствий;

Border\* border\_bottom – объекты границы;

Border\* border\_top;

Border\* border\_left;

Border\* border\_right;

FoodStorage\* foodStorage – указатель на харнилище пищи;

OreStorage\* oreStorage – указатель на харнилище руды;

void createHuman() – калссы для создания любого из объектов;

void createFoodObject(FoodColor foodColor, int pointsToCollectFoodObject);

void createOreObject(OreColor oreColor, int pointsToCollectOreObject);

void createObstacleObject(ObstacleColor obstacleColor, int width, int hight);

void DeleteHuman(Human\* humanToDelete) – классы для удаления любого из объектов;

void DeleteDepletedFoodObject(Food\* depletedFoodObject);

void DeleteDepletedOreObject(Ore\* depletedFoodObject);

void DeleteObstacleObject(Obstacle \*obstacleObject);

public slots:

void slotCheckFacedObject(QGraphicsItem \*facedObject, Human \*humanSender, int foodRequest, int oreRequest, bool isMovingToStorage) – слот для обработки запроса о распознавании встреченного сборщиком объекта;

**Human** – класс реализующий сборщика.

**Класс содержит:**

public:

void collectFood(int howMuchFood) – функция сбора еды;

void collectOre(int howMuchOre) – функция сбора руды;

void checkForCollision() - проверка коллизии на спавне;

void DoCollision() – выполнение столкновения;

void DoCollision(WhatSideCollisionIsFrom sideOFCollision) – перегруженная функция столкновения, реализует отражающее столкновение;

void unloadPackageOfFood() – разгрузка рюкзака с едой;

void unloadPackageOfOre()– разгрузка рюкзака с рудой;

void turnToNeededStorage(Storage\* storage) – поворот к хранилищу;

QPointF getLocation() {return location;} – получение позиции сборщика;

void UnloadEverything() - команда для немедленной доставке всего, что есть;

int GetPackageOfFood() {return packageOfFood;} – получить количество ресурсов пищи;

int GetPackageOfOre() {return packageOfOre;} – получить количество ресурсов руды;

int packageOfFood; - количество ресурсов пищи;

int packageOfOre; - количество ресурсов руды;

protected:

int startX; - начальная точка;

int startY;

qreal angle; - угол направдения;

qreal speed; - скорость;

QPointF location; - позиция на карте;

bool itemIsCollidingOnSpawn; - проверка столкновения на появлении;

bool isMovingToStorage; - в режиме доставки?;

bool isMovingToStorageNotFull; - в режиме срочной достаки?;

FoodStorage\* localFoodStorage; - ссылка на хранилище пищи;

OreStorage\* localOreStorage; - ссылка на хранилище руды;

QRectF boundingRect() const; - метод для определения коллизии;

void paint(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget); - метод прорисовки;

public slots:

void advance(int phase); - метод отвечающий за движение;

signals:

void signalCheckItem(QGraphicsItem \*item, Human \*humanSender, int foodRequest, int oreRequest, bool isMovingToStorage); - метод, запрашивающий информацию о встреченном объекте;

**CollectibleObject** – класс реализующий основу для всех собираемых предметов. От него реализуются классы ресурса пищи и руды. В наследуемых классах лишь переопределяется метод прорисовки и задания коллизии, которые наследуются еще от QGraphicsItem.

**Класс содержит:**

public:

bool getCollected(int &howMuchToCollect\_ref); – метод отвечающий за сбор ресурсов с объекта, в параметре возвращает количество собранных ресурсов;

void checkForCollision(); – проверка коллизии на появлении;

protected:

int startX; – начальная позиция;

int startY;

int pointsToCollect; количество единиц ресурса для сбора (до 300);

bool itemIsCollidingOnSpawn; - сигнал о том, что объект сталкивается на появлении;

virtual QRectF boundingRect() const = 0;

virtual void paint(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget) = 0;

**Obstacle** – класс реализующий препятствие, суть которого в изменении траектории движения сборщика в режиме поиска. Игнорируется сборщиком, когда тот в режиме доставки.

**Класс содержит**:

public:

void checkForCollision();– проверка коллизии на появлении;

WhatSideCollisionIsFrom getSideOfCollision(QPointF collidedItemLocation); - метод определяет с какой стороны сталкивается сборщик с препятствием и отправляет через главное окно информацию ему об этом;

protected:

ObstacleColor obstacleColor;

bool itemIsCollidingOnSpawn; // check if item has a place to spawn

int width; - ширина;

int height; - высота;

QPointF location – позиция на карте;

QRectF boundingRect() const;

void paint(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget);

**Storage** – виртуальный класс, реализующий основу для классов хранилища руды и пищи. В последних лишь переопределяется метод отрисовки и границы коллизии. Хранилище может содержать до 1500 единиц ресурса.

**Класс содержит**:

public:

bool GetResources(int howMuchResources); - получение хранилищем ресурсов;

void MakeEmpty() { prepareGeometryChange(); quantityOfResources = 0; }; - опустошение хранилище, используется при очистки карты;

protected:

int quantityOfResources; - общее количество ресурсов в хранилище (до 1500);

virtual QRectF boundingRect() const = 0;

virtual void paint(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget) = 0;

# Заключение

Была разработана программа, которая обладает следующим функционалом:

* + Демонстрация симуляции сбора ресурсов большим количество юнитов и доставка собранных ресурсов на точки сбора.

Возможны дальнейшие усовершенствования программы:

* Улучшение графической составляющей;
* Постройка динамических маршрутов, избегающих препятствия;
* Увеличение типов собираемых предметов, наследуемых от класса собираемого предмета;
* Добавление новых типов препятствий;
* Взаимодействие между сборщиками;

# Список использованных источников

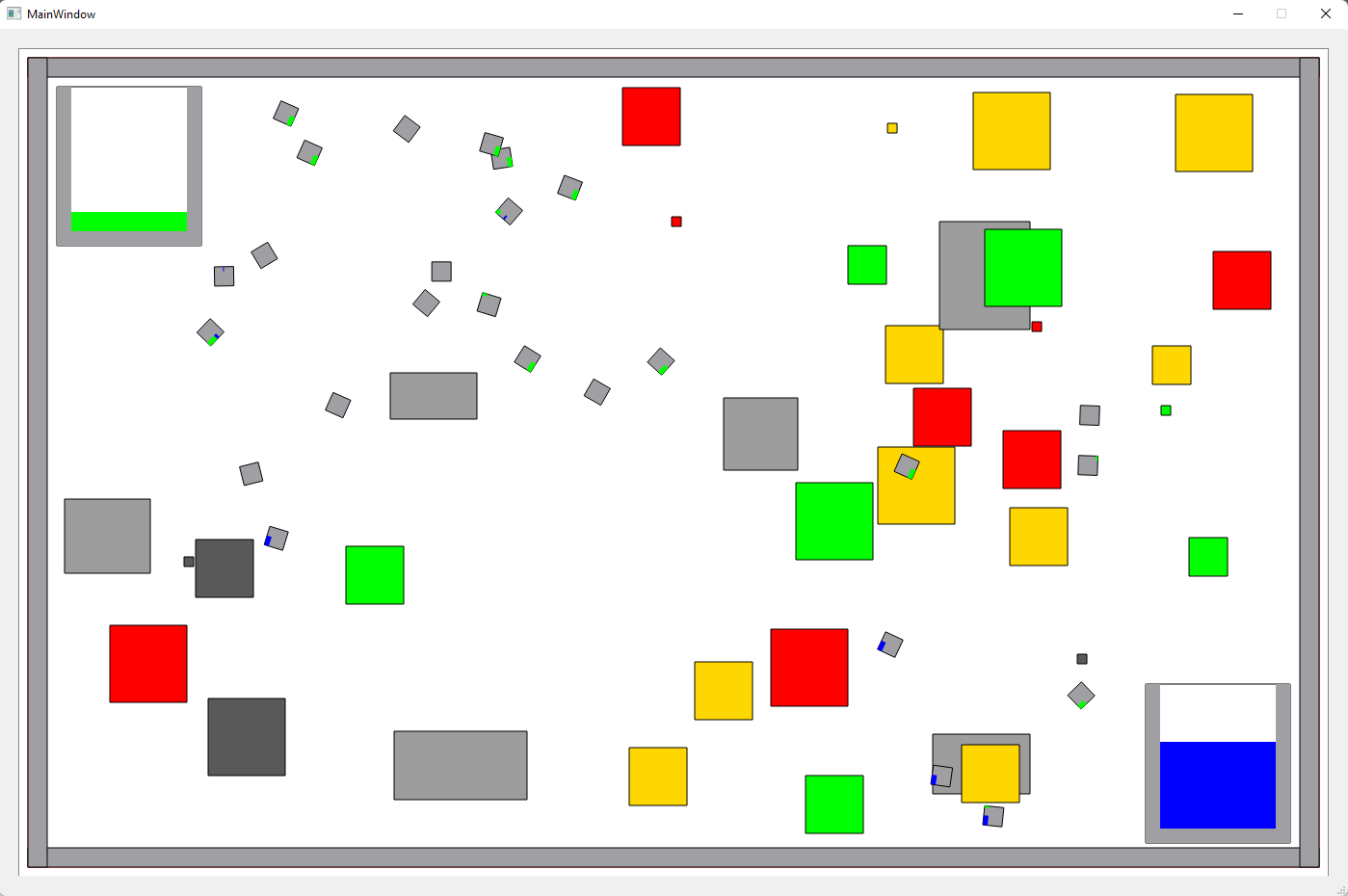
1. Пахомов Б.И. C/C++ и MS Visual C++ 2010 для начинающих. – СПб.: БХВ- Петербург, 2011. – 736с.
2. Википедия по Qt – Режим доступа: https://wiki.archlinux.org/title/Qt\_(Русский) и https://wiki.qt.io доступ свободный на каждый из сайтов.
3. Сайт https://forum.qt.io с обширной информацией о данной программе.
4. Сайт https://programforyou.ru/block-diagram-redactor для построения диограмм.

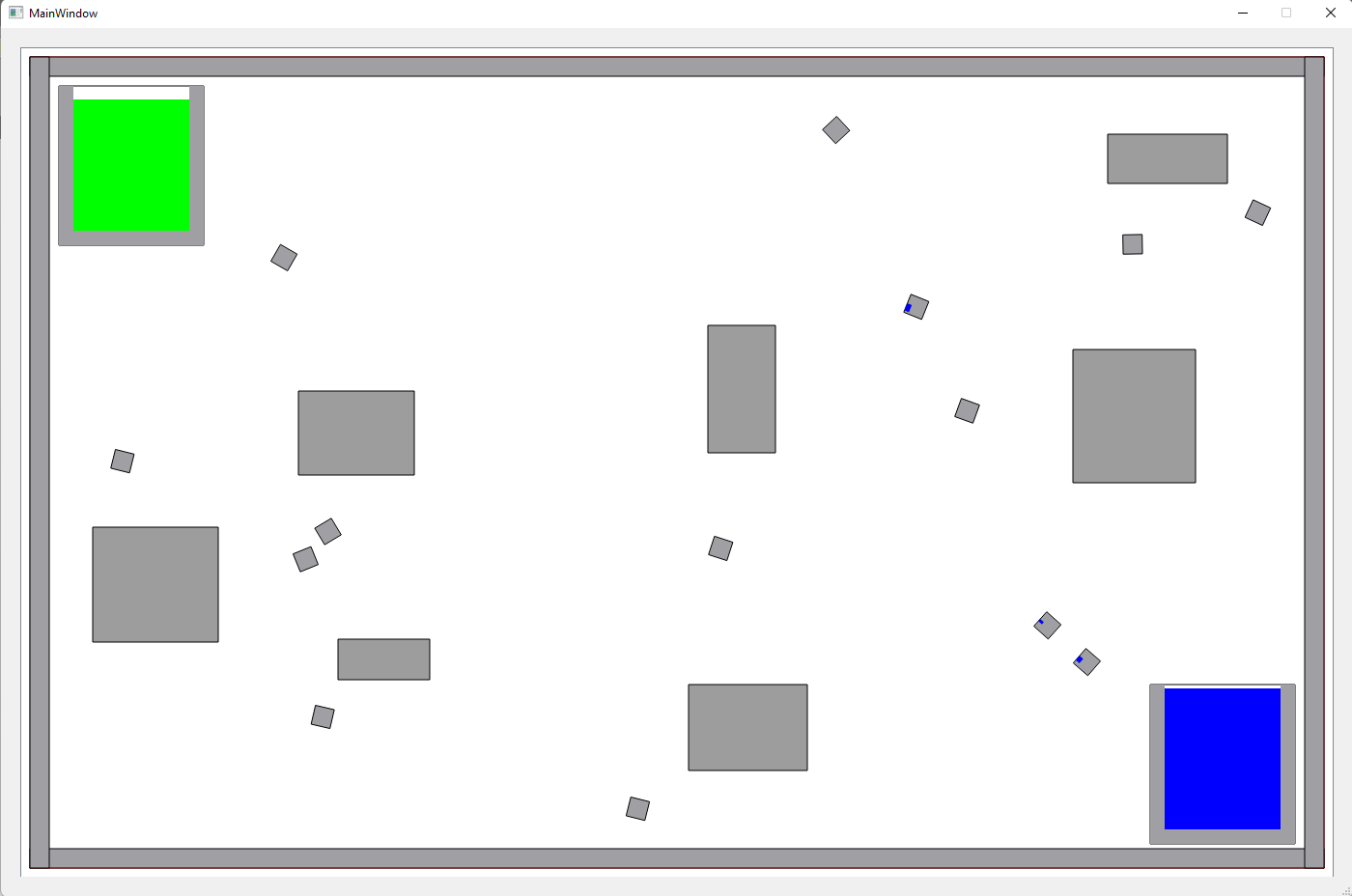
Приложение A. Снимки экранных форм   
пользовательского интерфейса

При загрузке программы появляется пустая карта с двумя хранилищами: для пищи в верхнем левом и для руды в нижнем правом.



Далее клавишей «пробел» создаем доставщиков, «1» создаем ресурсы руды, «2» создаем ресурсы пищи, «3» создаем препятствия.

Сборщики начали собирать ресурсы, и постепенно заполняют хранилища.



Со временем сборщики зачистят карту и на ней останутся лишь заполненные хранилища и препятствия.

Приложение Б. Исходный код.

**MainWindow.h:**

#ifndef MAINWINDOW\_H

#define **MAINWINDOW\_H**

#include <QMainWindow>

#include <QGraphicsScene>

#include <QtCore>

#include <QtGui>

#include <QLabel>>

#include <QKeyEvent>

#include "human.h"

#include "obstacle.h"

#include "ore.h"

#include "food.h"

#include "border.h"

#include "foodstorage.h"

#include "orestorage.h"

QT\_BEGIN\_NAMESPACE

*namespace* **Ui** { *class* **MainWindow**; }

QT\_END\_NAMESPACE

*class* **MainWindow** : *public* QMainWindow

{

Q\_OBJECT

*public*:

**MainWindow**(QWidget \***parent** = *nullptr*);

~**MainWindow**();

*private*:

Ui::MainWindow \***ui**;

QGraphicsScene \***scene**;

QTimer\* **timer**;

Human\* **human**;

QList<Human\*> **humans**;

QList<Food\*> **foodObjects**;

QList<Ore\*> **oreObjects**;

QList<Obstacle\*> **obstacleObjects**;

Border\* **border\_bottom**;

Border\* **border\_top**;

Border\* **border\_left**;

Border\* **border\_right**;

FoodStorage\* **foodStorage**;

OreStorage\* **oreStorage**;

int timerInterval;

void ***keyPressEvent***(QKeyEvent \***e**) *override*;

void **createHuman**();

void **createFoodObject**(FoodColor **foodColor**, int **pointsToCollectFoodObject**);

void **createOreObject**(OreColor **oreColor**, int **pointsToCollectOreObject**);

void **createObstacleObject**(ObstacleColor **obstacleColor**, int **width**, int **hight**);

void **DeleteHuman**(Human\* **humanToDelete**);

void **DeleteDepletedFoodObject**(Food\* **depletedFoodObject**);

void **DeleteDepletedOreObject**(Ore\* **depletedFoodObject**);

void **DeleteObstacleObject**(Obstacle \***obstacleObject**);

*public* slots:

void **slotCheckFacedObject**(QGraphicsItem \***facedObject**, Human \***humanSender**, int **foodRequest**, int **oreRequest**, bool **isMovingToStorage**);

void **slotDebug**(int **a**, int **b**, int **c**);

};

#endif *//* *MAINWINDOW\_H*

**MainWindow.cpp:**

#include "mainwindow.h"

#include "ui\_mainwindow.h"

MainWindow::**MainWindow**(QWidget \***parent**)

: QMainWindow(*parent*)

, ui(*new* Ui::MainWindow)

{

srand(time(0));

ui->setupUi(*this*);

*this*->resize(1400,900);

*this*->setFixedSize(1400,900);

scene = *new* QGraphicsScene();

ui->graphicsView->setScene(*scene*);

ui->graphicsView->setRenderHint(QPainter::*Antialiasing*);

ui->graphicsView->setVerticalScrollBarPolicy(Qt::*ScrollBarAlwaysOff*);

ui->graphicsView->setHorizontalScrollBarPolicy(Qt::*ScrollBarAlwaysOff*);

scene->setSceneRect(-670,-420,1340,840);

QPen **borderPen**(Qt::*red*);

QLineF **TopLine**(scene->sceneRect().topLeft(), scene->sceneRect().topRight());

QLineF **LeftLine**(scene->sceneRect().topLeft(), scene->sceneRect().bottomLeft());

QLineF **RightLine**(scene->sceneRect().topRight(), scene->sceneRect().bottomRight());

QLineF **BottomLine**(scene->sceneRect().bottomLeft(), scene->sceneRect().bottomRight());

scene->addLine(TopLine, borderPen);

scene->addLine(LeftLine, borderPen);

scene->addLine(RightLine, borderPen);

scene->addLine(BottomLine, borderPen);

border\_bottom = *new* Border(BorderSide::*BOTTOM\_BORDER*);

border\_top = *new* Border(BorderSide::*TOP\_BORDER*);

border\_left = *new* Border(BorderSide::*LEFT\_BORDER*);

border\_right = *new* Border(BorderSide::*RIGHT\_BORDER*);

scene->addItem(*border\_bottom*);

scene->addItem(*border\_top*);

scene->addItem(*border\_left*);

scene->addItem(*border\_right*);

foodStorage = *new* FoodStorage();

oreStorage = *new* OreStorage();

scene->addItem(*foodStorage*);

scene->addItem(*oreStorage*);

timer = *new* QTimer(*this*);

connect(timer, SIGNAL(timeout()), scene, SLOT(advance()));

timer->start(2);

}

MainWindow::~**MainWindow**()

{

*delete* ui;

}

void MainWindow::***keyPressEvent***(QKeyEvent \***e**)

{

int **tempRand**;

*if* (e->key() == Qt::*Key\_Space*)

{

createHuman();

};

*if* (e->key() == Qt::*Key\_7*)

{

*if* (timerInterval < 10)

{

timerInterval++;

timer->setInterval(timerInterval);

}

};

*if* (e->key() == Qt::*Key\_6*)

{

*if* (timerInterval > 1)

{

timerInterval--;

timer->setInterval(timerInterval);

}

};

*if* (e->key() == Qt::*Key\_5*)

{

foreach(Human\* **humanToUnload**, *humans*)

{

humanToUnload->UnloadEverything();

}

};

*if* (e->key() == Qt::*Key\_4*)

{

foreach(Food\* **foodObjectToDelete**, *foodObjects*)

{

DeleteDepletedFoodObject(foodObjectToDelete);

}

foreach(Ore\* **oreObjectToDelete**, *oreObjects*)

{

DeleteDepletedOreObject(oreObjectToDelete);

}

foreach(Obstacle\* **obstacleObjectToDelete**, *obstacleObjects*)

{

DeleteObstacleObject(obstacleObjectToDelete);

}

foreach(Human\* **humanToDelete**, *humans*)

{

DeleteHuman(humanToDelete);

}

foodStorage->MakeEmpty();

oreStorage->MakeEmpty();

};

*if* (e->key() == Qt::*Key\_3*)

{

tempRand = rand() % (2);

*switch* (tempRand)

{

*case* 1:

createObstacleObject(ObstacleColor::*ROCK*, rand() % (100) + 40, rand() % (100) + 40);

*break*;

*case* 2:

createObstacleObject(ObstacleColor::*WOOD*, rand() % (100) + 40, rand() % (100) + 40);

*break*;

}

};

*if* (e->key() == Qt::*Key\_2*)

{

tempRand = rand() % (3);

*switch* (tempRand)

{

*case* 1:

createFoodObject(FoodColor::*GREEN*, rand() % (200) + 100);

*break*;

*case* 2:

createFoodObject(FoodColor::*RED*, rand() % (200) + 100);

*break*;

*case* 3:

createFoodObject(FoodColor::*YELLOW*, rand() % (200) + 100);

*break*;

}

};

*if* (e->key() == Qt::*Key\_1*)

{

tempRand = rand() % (3);

*switch* (tempRand)

{

*case* 1:

createOreObject(OreColor::*GOLDEN*, rand() % (300));

*break*;

*case* 2:

createOreObject(OreColor::*IRON*, rand() % (300));

*break*;

*case* 3:

createOreObject(OreColor::*COPPER*, rand() % (300));

*break*;

}

};

}

void MainWindow::**createHuman**()

{

human = *new* Human(*foodStorage*, *oreStorage*);

scene->addItem(*human*);

humans.append(*human*);

human->checkForCollision();

connect(human, &Human::signalCheckItem, *this*, &MainWindow::slotCheckFacedObject);

connect(human, &Human::signalDebug, *this*, &MainWindow::slotDebug);

}

void MainWindow::**createFoodObject**(FoodColor **foodColor**, int **pointsToCollectFoodObject** = 180)

{

Food \***foodObject** = *new* Food(foodColor, pointsToCollectFoodObject);

scene->addItem(*foodObject*);

foodObjects.append(*foodObject*);

foodObject->checkForCollision();

}

void MainWindow::**createOreObject**(OreColor **oreColor**, int **pointsToCollectFoodObject** = 180)

{

Ore \***oreObject** = *new* Ore(oreColor, pointsToCollectFoodObject);

scene->addItem(*oreObject*);

oreObjects.append(*oreObject*);

oreObject->checkForCollision();

}

void MainWindow::**createObstacleObject**(ObstacleColor **obstacleColor**, int **width**, int **hight**)

{

Obstacle \***obstacleObject** = *new* Obstacle(obstacleColor, width, hight);

scene->addItem(*obstacleObject*);

obstacleObjects.append(*obstacleObject*);

obstacleObject->checkForCollision();

}

void MainWindow::**slotCheckFacedObject**(QGraphicsItem \***facedObject**, Human\* **humanSender**, int **foodRequest**, int **oreRequest**, bool **isMovingToStorage**)

{

*if*(!isMovingToStorage)

{

foreach(Food\* **possibleFoodObject**, *foodObjects*)

{

*if* (possibleFoodObject == facedObject)

{

*if* (possibleFoodObject->getCollected(*foodRequest*)) *//* *true* *if* *depleted*

{

DeleteDepletedFoodObject(possibleFoodObject);

}

humanSender->collectFood(foodRequest);

}

}

foreach(Ore\* **possibleOreObject**, *oreObjects*)

{

*if* (possibleOreObject == facedObject)

{

*if* (possibleOreObject->getCollected(*oreRequest*)) *//* *true* *if* *depleted*

{

DeleteDepletedOreObject(possibleOreObject);

}

humanSender->collectOre(oreRequest);

}

}

foreach(Obstacle\* **possibleObstacleObject**, *obstacleObjects*)

{

*if* (possibleObstacleObject == facedObject)

{

humanSender->DoCollision(possibleObstacleObject->getSideOfCollision(humanSender->getLocation()));

}

}

*if* (facedObject == foodStorage || facedObject == oreStorage)

humanSender->DoCollision();

}

*else*

{

*if* (facedObject == foodStorage)

{

foodStorage->GetResources(humanSender->GetPackageOfFood());

humanSender->unloadPackageOfFood();

}

*if* (facedObject == oreStorage)

{

oreStorage->GetResources(humanSender->GetPackageOfOre());

humanSender->unloadPackageOfOre();

}

}

*if* (facedObject == border\_bottom)

humanSender->DoCollision(WhatSideCollisionIsFrom::*TOP*);

*if* (facedObject == border\_top)

humanSender->DoCollision(WhatSideCollisionIsFrom::*BOTTOM*);

*if* (facedObject == border\_left)

humanSender->DoCollision(WhatSideCollisionIsFrom::*RIGHT*);

*if* (facedObject == border\_right)

humanSender->DoCollision(WhatSideCollisionIsFrom::*LEFT*);

}

void MainWindow::**slotDebug**(int **a**, int **b**, int **c**)

{

ui->label->setText(QString::number(a));

ui->label\_2->setText(QString::number(b));

ui->label\_3->setText(QString::number(c));

}

void MainWindow::**DeleteDepletedFoodObject**(Food \***depletedFoodObject**)

{

scene->removeItem(*depletedFoodObject*);

foodObjects.removeOne(*depletedFoodObject*);

}

void MainWindow::**DeleteDepletedOreObject**(Ore \***depletedOreObject**)

{

scene->removeItem(*depletedOreObject*);

oreObjects.removeOne(*depletedOreObject*);

}

void MainWindow::**DeleteObstacleObject**(Obstacle \***obstacleObject**)

{

scene->removeItem(*obstacleObject*);

obstacleObjects.removeOne(*obstacleObject*);

}

void MainWindow::**DeleteHuman**(Human \***humanToDelete**)

{

scene->removeItem(*humanToDelete*);

humans.removeOne(*humanToDelete*);

}

**Human.h:**

#ifndef HUMAN\_H

#define **HUMAN\_H**

#include <QPainter>

#include <QGraphicsItem>

#include <QGraphicsScene>

#include <QObject>

#include "food.h"

#include "obstacle.h"

#include "foodstorage.h"

#include "orestorage.h"

#include <cmath>

*class* **Human** : *public* QObject, *public* QGraphicsItem

{

Q\_OBJECT

*public*:

**Human**(FoodStorage\* **foodStorage**, OreStorage\* **oreStorage**);

~**Human**() {};

void **collectFood**(int **howMuchFood**);

void **collectOre**(int **howMuchOre**);

void **checkForCollision**();

void **DoCollision**();

void **DoCollision**(WhatSideCollisionIsFrom **sideOFCollision**);

void **unloadPackageOfFood**();

void **unloadPackageOfOre**();

void **turnToNeededStorage**(Storage\* **storage**);

QPointF **getLocation**() {*return* location;}

void **UnloadEverything**();

int **GetPackageOfFood**() {*return* packageOfFood;}

int **GetPackageOfOre**() {*return* packageOfOre;}

int **packageOfFood**; *//* *from* *0* *to* *100*

int **packageOfOre**; *//* *from* *0* *to* *100*

*protected*:

int **startX**;

int **startY**;

qreal **angle**;

qreal **speed**;

QPointF **location**;

bool **itemIsCollidingOnSpawn**; *//* *check* *if* *item* *has* *a* *place* *to* *spawn*

bool **isMovingToStorage**;

bool **isMovingToStorageNotFull**;

FoodStorage\* **localFoodStorage**;

OreStorage\* **localOreStorage**;

QRectF ***boundingRect***() *const*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**);

*public* slots:

void ***advance***(int **phase**);

signals:

void **signalCheckItem**(QGraphicsItem \***item**, Human \***humanSender**, int **foodRequest**, int **oreRequest**, bool **isMovingToStorage**);

void **signalDebug**(int **a**, int **b**, int **c**);

};

#endif *//* *HUMAN\_H*

**Human.cpp:**

#include "human.h"

Human::**Human**(FoodStorage\* **foodStorage**, OreStorage\* **oreStorage**)

: QObject(), QGraphicsItem()

{

setFlag(*this*->*ItemIsMovable*);

srand(time(0));

*//random* *start* *rotation*

angle = (rand() % 360);

setRotation(angle);

*//set* *the* *speed*

speed = 1;

*//randon* *start* *position*

srand(time(0));

startX = (rand() % (670)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (420)) \* ((rand()%2 == 1)?1:-1);

setPos(startX, startY);

packageOfFood = 0;

packageOfOre = 0;

*this*->localFoodStorage = foodStorage;

*this*->localOreStorage = oreStorage;

isMovingToStorage = *false*;

}

void Human::**collectFood**(int **howMuchFood**)

{

packageOfFood += howMuchFood;

*if* (packageOfFood >= 100)

{

packageOfFood = 100;

isMovingToStorage = *true*;

turnToNeededStorage(*localFoodStorage*);

}

*if* (isMovingToStorageNotFull == *true*)

turnToNeededStorage(*localFoodStorage*);

}

void Human::**collectOre**(int **howMuchOre**)

{

packageOfOre += howMuchOre;

*if* (packageOfOre >= 100)

{

packageOfOre = 100;

isMovingToStorage = *true*;

turnToNeededStorage(*localOreStorage*);

}

*if* (isMovingToStorageNotFull == *true*)

turnToNeededStorage(*localOreStorage*);

}

QRectF Human::***boundingRect***() *const*

{

*return* QRectF(-10, -10, 20, 20);

}

void Human::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

QBrush **brush**(Qt::*gray*);

painter->fillRect(rec, brush);

painter->drawRect(rec);

QRectF **lineRec\_food**(-10, 5, (int) packageOfFood/10, 5);

brush.setColor(Qt::*green*);

painter->setPen(Qt::*NoPen*);

brush.setStyle(Qt::*SolidPattern*);

painter->fillRect(lineRec\_food, brush);

QRectF **lineRec\_ore**(0, 5, (int) packageOfOre/10, 5);

brush.setColor(Qt::*blue*);

painter->setPen(Qt::*NoPen*);

brush.setStyle(Qt::*SolidPattern*);

painter->fillRect(lineRec\_ore, brush);

}

void Human::**unloadPackageOfFood**()

{

packageOfFood = 0;

isMovingToStorage = *false*;

isMovingToStorageNotFull = *false*;

}

void Human::**unloadPackageOfOre**()

{

packageOfOre = 0;

isMovingToStorage = *false*;

isMovingToStorageNotFull = *false*;

}

void Human::**turnToNeededStorage**(Storage \***storage**)

{

bool **isStorageFound** = *false*;

int **locAngle** = 0;

*do* {

QList<QGraphicsItem \*> **foundItems** = scene()->items(QPolygonF()

<< mapToScene(0, 0) << mapToScene(-1, -1700) << mapToScene(1, -1700));

foreach (QGraphicsItem \***item**, *foundItems*)

{

*if* (item == storage)

isStorageFound = *true*;

*else* {

setRotation(locAngle + 1);

locAngle = locAngle + 1;

}

}

} *while* (!isStorageFound);

}

void Human::**UnloadEverything**()

{

*if* (!isMovingToStorage)

{

*if* (packageOfFood != 0)

{

turnToNeededStorage(*localFoodStorage*);

isMovingToStorage = *true*;

isMovingToStorageNotFull = *true*;

}

*else* *if* (packageOfOre != 0 )

{

turnToNeededStorage(*localOreStorage*);

isMovingToStorage = *true*;

isMovingToStorageNotFull = *true*;

}

}

}

void Human::***advance***(int **phase**)

{

*if*(!phase) *return*;

setPos(mapToParent(0, -(speed)));

location = *this*->pos();

QList<QGraphicsItem \*> **foundItems** = scene()->items(QPolygonF()

<< mapToScene(0, 0) << mapToScene(-20, -20) << mapToScene(20, -20));

foreach (QGraphicsItem \***item**, *foundItems*)

{

*if* (item == *this*)

*continue*;

emit signalCheckItem(item, *this*, 100 - packageOfFood, 100 - packageOfOre, isMovingToStorage);

}

}

void Human::**checkForCollision**()

{

*do* {

*if*(scene()->collidingItems(*this*).isEmpty())

{

itemIsCollidingOnSpawn = *false*;

}

*else* {

itemIsCollidingOnSpawn = *true*;

startX = (rand() % (670)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (420)) \* ((rand()%2 == 1)?1:-1);

setPos(startX, startY);

}

} *while*(itemIsCollidingOnSpawn);

}

void Human::**DoCollision**(WhatSideCollisionIsFrom **sideOfCollision**)

{

*if* (isMovingToStorage)

{

collectFood(0);

collectOre(0);

}

*else*

*switch*(sideOfCollision)

{

*case* WhatSideCollisionIsFrom::*TOP*:

*if* (angle > 180)

angle = angle + (270 - angle) \* 2;

*else* angle = angle - (angle - 90) \* 2;

setRotation(angle);

*break*;

*case* WhatSideCollisionIsFrom::*BOTTOM*:

*if* (angle > 180)

angle = angle - (angle - 270) \* 2;

*else* angle = angle + (90 - angle) \* 2;

setRotation(angle);

*break*;

*case* WhatSideCollisionIsFrom::*RIGHT*:

*if* (angle > 90 && angle < 270)

angle = angle - (angle - 180) \* 2;

*else* angle = 0 + (360 - angle);

setRotation(angle);

*break*;

*case* WhatSideCollisionIsFrom::*LEFT*:

*if* (angle > 90 && angle < 270)

angle = angle + (180 - angle) \* 2;

*else* angle = 270 + (90 - angle);

setRotation(angle);

*break*;

}

}

void Human::**DoCollision**()

{

angle = (rand() % (180)) \* ((rand()%2 == 1)?1:-1);

setRotation(angle);

}

**ColllectibleObject.h:**

#ifndef COLLECTIBLEOBJECT\_H

#define **COLLECTIBLEOBJECT\_H**

#include <QGraphicsItem>

#include <QPainter>

#include <QGraphicsItem>

#include <QGraphicsScene>

#include <QObject>

*enum* *class* **TypeOfFacedObject**

{

***FOOD***,

***ORE***,

***OBSTACLE***

};

*class* **CollectibleObject** : *public* QObject, *public* QGraphicsItem

{

Q\_OBJECT

*public*:

**CollectibleObject**();

~**CollectibleObject**() {};

bool **getCollected**(int &**howMuchToCollect\_ref**); *//* *value* *of* *collected* *food* *will* *be* *returned* *in* *<howMuchToCollect>*

*//* *true* *if* *object* *is* *depleted*

void **checkForCollision**();

*protected*:

int **startX**;

int **startY**;

int **pointsToCollect**; *//* *from* *1* *to* *300*

bool **itemIsCollidingOnSpawn**; *//* *check* *if* *item* *has* *a* *place* *to* *spawn*

*virtual* QRectF ***boundingRect***() *const* = 0;

*virtual* void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) = 0;

};

#endif *//* *COLLECTIBLEOBJECT\_H*

**CollectibleObject.cpp:**

#include "collectibleobject.h"

CollectibleObject::**CollectibleObject**()

: QObject(), QGraphicsItem()

{

srand(time(0));

startX = (rand() % (600)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (400)) \* ((rand()%2 == 1)?1:-1);

}

bool CollectibleObject::**getCollected**(int &**howMuchToCollect\_ref**)

{

*if* (howMuchToCollect\_ref >= pointsToCollect)

{

howMuchToCollect\_ref = pointsToCollect;

pointsToCollect = 0;

prepareGeometryChange();

*return* *true*;

} *else*

{

pointsToCollect = pointsToCollect - howMuchToCollect\_ref;

prepareGeometryChange();

*return* *false*;

}

}

void CollectibleObject::**checkForCollision**()

{

*do* {

*if*(scene()->collidingItems(*this*).isEmpty())

{

itemIsCollidingOnSpawn = *false*;

}

*else* {

itemIsCollidingOnSpawn = *true*;

startX = (rand() % (600)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (400)) \* ((rand()%2 == 1)?1:-1);

setPos(startX, startY);

}

} *while*(itemIsCollidingOnSpawn);

}

**Storage.h:**

#ifndef STORAGE\_H

#define **STORAGE\_H**

#include <QGraphicsItem>

#include <QPainter>

#include <QGraphicsItem>

#include <QGraphicsScene>

#include <QObject>

*class* **Storage** : *public* QObject, *public* QGraphicsItem

{

*public*:

**Storage**();

~**Storage**() {};

bool **GetResources**(int **howMuchResources**); *//* *returns* *true* *if* *storage* *is* *full*

void **MakeEmpty**() { prepareGeometryChange(); quantityOfResources = 0; };

*protected*:

int **quantityOfResources**; *//* *from* *0* *to* *1500*

int **startX**;

int **startY**;

*virtual* QRectF ***boundingRect***() *const* = 0;

*virtual* void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) = 0;

};

#endif *//* *STORAGE\_H*

**Storage.cpp:**

#include "storage.h"

Storage::**Storage**()

{

quantityOfResources = 0;

}

bool Storage::**GetResources**(int **howMuchResources**)

{

quantityOfResources += howMuchResources;

*if* (quantityOfResources >= 1500)

{

quantityOfResources = 0;

prepareGeometryChange();

*return* *true*;

}

*else*

{

prepareGeometryChange();

*return* *false*;

}

}

**FoodStorage.h:**

#ifndef FOODSTORAGE\_H

#define **FOODSTORAGE\_H**

#include "storage.h"

*class* **FoodStorage** : *public* Storage

{

*public*:

**FoodStorage**();

~**FoodStorage**() {};

*private*:

QRectF ***boundingRect***() *const* *override*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) *override*;

};

#endif *//* *FOODSTORAGE\_H*

**FoodStorage.cpp:**

#include "foodstorage.h"

FoodStorage::**FoodStorage**()

{

startX = -640;

startY = -390;

setPos(startX,startY);

}

QRectF FoodStorage::***boundingRect***() *const*

{

*return* QRectF(0, 0, 150, 165);

}

void FoodStorage::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

painter->drawRect(rec);

QBrush **brush\_1**;

brush\_1.setColor(Qt::*gray*);

QRectF **LeftWall**(0, 0, 15, 150);

painter->setPen(Qt::*NoPen*);

brush\_1.setStyle(Qt::*SolidPattern*);

painter->fillRect(LeftWall, brush\_1);

QBrush **brush\_2**;

brush\_2.setColor(Qt::*gray*);

QRectF **RightWall**(135, 0, 15, 150);

painter->setPen(Qt::*NoPen*);

brush\_2.setStyle(Qt::*SolidPattern*);

painter->fillRect(RightWall, brush\_2);

QBrush **brush\_3**;

brush\_3.setColor(Qt::*gray*);

QRectF **BottomWall**(0, 150, 150, 15);

painter->setPen(Qt::*NoPen*);

brush\_3.setStyle(Qt::*SolidPattern*);

painter->fillRect(BottomWall, brush\_3);

QBrush **brush\_4**;

brush\_4.setColor(Qt::*green*);

QRectF **Resource**(15, 150 - (int) quantityOfResources/10, 120, (int) quantityOfResources/10); *//* *width* *height*

painter->setPen(Qt::*NoPen*);

brush\_4.setStyle(Qt::*SolidPattern*);

painter->fillRect(Resource, brush\_4);

}

**OreStorage.h:**

#ifndef ORESTORAGE\_H

#define **ORESTORAGE\_H**

#include "storage.h"

*class* **OreStorage** : *public* Storage

{

*public*:

**OreStorage**();

~**OreStorage**() {};

*private*:

QRectF ***boundingRect***() *const* *override*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) *override*;

};

#endif *//* *ORESTORAGE\_H*

**OreStorage.cpp:**

#include "orestorage.h"

OreStorage::**OreStorage**()

{

startX = 490; *//650*

startY = 230; *//400*

setPos(startX,startY);

}

QRectF OreStorage::***boundingRect***() *const*

{

*return* QRectF(0, 0, 150, 165);

}

void OreStorage::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

painter->drawRect(rec);

QBrush **brush\_1**;

brush\_1.setColor(Qt::*gray*);

QRectF **LeftWall**(0, 0, 15, 150);

painter->setPen(Qt::*NoPen*);

brush\_1.setStyle(Qt::*SolidPattern*);

painter->fillRect(LeftWall, brush\_1);

QBrush **brush\_2**;

brush\_2.setColor(Qt::*gray*);

QRectF **RightWall**(135, 0, 15, 150);

painter->setPen(Qt::*NoPen*);

brush\_2.setStyle(Qt::*SolidPattern*);

painter->fillRect(RightWall, brush\_2);

QBrush **brush\_3**;

brush\_3.setColor(Qt::*gray*);

QRectF **BottomWall**(0, 150, 150, 15);

painter->setPen(Qt::*NoPen*);

brush\_3.setStyle(Qt::*SolidPattern*);

painter->fillRect(BottomWall, brush\_3);

QBrush **brush\_4**;

brush\_4.setColor(Qt::*blue*);

QRectF **Resource**(15, 150 - (int) quantityOfResources/10, 120, (int) quantityOfResources/10);

painter->setPen(Qt::*NoPen*);

brush\_4.setStyle(Qt::*SolidPattern*);

painter->fillRect(Resource, brush\_4);

}

**Food.h:**

#ifndef FOOD\_H

#define **FOOD\_H**

#include "collectibleobject.h"

*enum* *class* **FoodColor**

{

***GREEN***,

***RED***,

***YELLOW***

};

*class* **Food** : *public* CollectibleObject

{

*public*:

**Food**(FoodColor **foodColor**, int **pointsToCollect**);

~**Food**() {};

*protected*:

FoodColor **foodColor**;

QRectF ***boundingRect***() *const* *override*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) *override*;

};

#endif *//* *FOOD\_H*

**Food.cpp:**

#include "food.h"

Food::**Food**(FoodColor **foodColor**, int **pointsToCollect** = 180)

{

*this*->foodColor = foodColor;

*if* (pointsToCollect > 300)

*this*->pointsToCollect = 300;

*else* *if* (pointsToCollect < 1)

*this*->pointsToCollect = 1;

*else* *this*->pointsToCollect = pointsToCollect;

setFlag(*this*->*ItemIsMovable*);

setPos(startX, startY);

}

QRectF Food::***boundingRect***() *const*

{

*if* (pointsToCollect < 300)

*if* (pointsToCollect < 200)

*if* (pointsToCollect < 100)

*if* (pointsToCollect < 50)

*if* (pointsToCollect < 20)

*if* (pointsToCollect == 0)

*return* QRectF(0,0,0,0);

*else* *return* QRectF(0,0,10,10);

*else* *return* QRectF(0,0,25,25);

*else* *return* QRectF(0,0,40,40);

*else* *return* QRectF(0,0,60,60);

*else* *return* QRectF(0,0,80,80);

*else* *return* QRectF(0,0,100,100);

}

void Food::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

QBrush **brush**;

*switch*(foodColor)

{

*case* FoodColor::*GREEN*:

brush.setColor(Qt::*green*);

brush.setStyle(Qt::*SolidPattern*);

*break*;

*case* FoodColor::*RED*:

brush.setColor(Qt::*red*);

brush.setStyle(Qt::*SolidPattern*);

*break*;

*case* FoodColor::*YELLOW*:

brush.setColor(Qt::*yellow*);

brush.setStyle(Qt::*SolidPattern*);

*break*;

}

painter->fillRect(rec, brush);

painter->drawRect(rec);

}

**Ore.h:**

#ifndef ORE\_H

#define **ORE\_H**

#include "collectibleobject.h"

*enum* *class* **OreColor**

{

***GOLDEN***,

***IRON***,

***COPPER***

};

*class* **Ore** : *public* CollectibleObject

{

*public*:

**Ore**(OreColor **oreColor**, int **pointsToCollect**);

~**Ore**() {};

*protected*:

OreColor **oreColor**;

QRectF ***boundingRect***() *const* *override*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**) *override*;

};

#endif *//* *ORE\_H*

**Ore.cpp:**

#include "ore.h"

Ore::**Ore**(OreColor **oreColor**, int **pointsToCollect** = 180)

{

*this*->oreColor = oreColor;

*if* (pointsToCollect > 300)

*this*->pointsToCollect = 300;

*else* *if* (pointsToCollect < 1)

*this*->pointsToCollect = 1;

*else* *this*->pointsToCollect = pointsToCollect;

setFlag(*this*->*ItemIsMovable*);

setPos(startX, startY);

}

QRectF Ore::***boundingRect***() *const*

{

*if* (pointsToCollect < 300)

*if* (pointsToCollect < 200)

*if* (pointsToCollect < 100)

*if* (pointsToCollect < 50)

*if* (pointsToCollect < 20)

*if* (pointsToCollect == 0)

*return* QRectF(0,0,0,0);

*else* *return* QRectF(0,0,10,10);

*else* *return* QRectF(0,0,25,25);

*else* *return* QRectF(0,0,40,40);

*else* *return* QRectF(0,0,60,60);

*else* *return* QRectF(0,0,80,80);

*else* *return* QRectF(0,0,100,100);

}

void Ore::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

QBrush **brush**;

*switch*(oreColor)

{

*case* OreColor::*GOLDEN*:

brush.setColor(QColorConstants::Svg::gold);

brush.setStyle(Qt::*SolidPattern*);

*break*;

*case* OreColor::*IRON*:

brush.setColor(QColor(89, 89, 89));

brush.setStyle(Qt::*SolidPattern*);

*break*;

*case* OreColor::*COPPER*:

brush.setColor(QColor(94, 98, 58));

brush.setStyle(Qt::*SolidPattern*);

*break*;

}

painter->fillRect(rec, brush);

painter->drawRect(rec);

}

**Border.h:**

#ifndef BORDER\_H

#define **BORDER\_H**

#include <QPainter>

#include <QGraphicsItem>

#include <QGraphicsScene>

#include <QObject>

*enum* *class* **BorderSide**

{

***LEFT\_BORDER***,

***RIGHT\_BORDER***,

***TOP\_BORDER***,

***BOTTOM\_BORDER***

};

*class* **Border** : *public* QObject, *public* QGraphicsItem

{

*public*:

**Border**(BorderSide **borderSide**);

~**Border**() {};

QRectF ***boundingRect***() *const*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**);

*private*:

BorderSide **border**;

};

#endif *//* *BORDER\_H*

**Border.cpp:**

#include "border.h"

Border::Border(BorderSide borderSide) : QObject(), QGraphicsItem()

{

border = borderSide;

*switch* (border)

{

*case* BorderSide::BOTTOM\_BORDER:

setPos(-670, 400);

*break*;

*case* BorderSide::TOP\_BORDER:

setPos(-670, -420);

*break*;

*case* BorderSide::LEFT\_BORDER:

setPos(-670, -420);

*break*;

*case* BorderSide::RIGHT\_BORDER:

setPos(650, -420);

*break*;

}

}

QRectF Border::boundingRect() *const*

{

*switch* (border)

{

*case* BorderSide::BOTTOM\_BORDER:

*case* BorderSide::TOP\_BORDER:

*return* QRectF(0,0, 1340, 20);

*break*;

*case* BorderSide::LEFT\_BORDER:

*case* BorderSide::RIGHT\_BORDER:

*return* QRectF(0,0, 20, 840);

*break*;

}

}

void Border::paint(QPainter \*painter, *const* QStyleOptionGraphicsItem \*option, QWidget \*widget)

{

QRectF rec = boundingRect();

QBrush brush;

brush.setColor(Qt::gray);

brush.setStyle(Qt::SolidPattern);

painter->fillRect(rec, brush);

painter->drawRect(rec);

}

**Obstacle.h:**

#ifndef OBSTACLE\_H

#define **OBSTACLE\_H**

#include <QGraphicsItem>

#include <QGraphicsScene>

#include <QPainter>

#include <QObject>

*enum* *class* **ObstacleColor**

{

***ROCK***,

***WOOD***

};

*enum* *class* **WhatSideCollisionIsFrom**

{

***TOP***,

***BOTTOM***,

***RIGHT***,

***LEFT***

};

*class* **Obstacle** : *public* QObject, *public* QGraphicsItem

{

Q\_OBJECT

*public*:

**Obstacle**(ObstacleColor **obstacleColor**, int **width**, int **height**);

~**Obstacle**() {};

void **checkForCollision**();

WhatSideCollisionIsFrom **getSideOfCollision**(QPointF **collidedItemLocation**);

*protected*:

ObstacleColor **obstacleColor**;

bool **itemIsCollidingOnSpawn**; *//* *check* *if* *item* *has* *a* *place* *to* *spawn*

int **width**;

int **height**;

int **startX**;

int **startY**;

QPointF **location**;

QRectF ***boundingRect***() *const*;

void ***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**);

};

#endif *//* *OBSTACLE\_H*

**Obstacle.cpp:**

#include "obstacle.h"

Obstacle::**Obstacle**(ObstacleColor **obstacleColor**, int **width**, int **height**) : QObject(), QGraphicsItem()

{

*this*->obstacleColor = obstacleColor;

*this*->width = width;

*this*->height = height;

setFlag(*this*->*ItemIsMovable*);

srand(time(0));

startX = (rand() % (600)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (400)) \* ((rand()%2 == 1)?1:-1);

}

QRectF Obstacle::***boundingRect***() *const*

{

*return* QRectF(0, 0, *this*->width, *this*->height);

}

void Obstacle::***paint***(QPainter \***painter**, *const* QStyleOptionGraphicsItem \***option**, QWidget \***widget**)

{

QRectF **rec** = *boundingRect*();

QBrush **brush**;

*switch*(obstacleColor)

{

*case* ObstacleColor::*ROCK*:

brush.setColor(QColor(157, 157, 157));

brush.setStyle(Qt::*SolidPattern*);

*break*;

*case* ObstacleColor::*WOOD*:

brush.setColor(QColor(72, 64, 39));

brush.setStyle(Qt::*SolidPattern*);

*break*;

}

painter->fillRect(rec, brush);

painter->drawRect(rec);

}

void Obstacle::**checkForCollision**()

{

*do* {

*if*(scene()->collidingItems(*this*).isEmpty())

{

itemIsCollidingOnSpawn = *false*;

}

*else* {

itemIsCollidingOnSpawn = *true*;

startX = (rand() % (670)) \* ((rand()%2 == 1)?1:-1);

startY = (rand() % (420)) \* ((rand()%2 == 1)?1:-1);

setPos(startX, startY);

}

} *while*(itemIsCollidingOnSpawn);

}

WhatSideCollisionIsFrom Obstacle::**getSideOfCollision**(QPointF **collidedItemLocation**)

{

location = *this*->pos();

*if* (collidedItemLocation.y() > location.y())

*if* (collidedItemLocation.x() > location.x())

*if* (collidedItemLocation.x() < location.x() + width)

*return* WhatSideCollisionIsFrom::*BOTTOM*;

*else* *return* WhatSideCollisionIsFrom::*RIGHT*;

*else* *return* WhatSideCollisionIsFrom::*LEFT*;

*else* *return* WhatSideCollisionIsFrom::*TOP*;

}