Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«МАТИ - Российский государственный технологический университет имени К.Э.Циолковского» (МАТИ)

КАФЕДРА: «Моделирование систем и информационные технологии»

РЕЦЕНЗЕНТ _____ ЗАВ. КАФЕДРОЙ _____

“_____”_____ 2015 г. “_____”_____ 2015 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

по направлению 230100.62 «Информатика и вычислительная техника»

ТЕМА: Разработка компьютерной игры Pacman с элементами искусственного интеллекта на платформе Qt

Студент _____ А. С. Сапин
подпись расшифровка подписи

Руководитель _____ В. В. Лидовский

Консультант _____ А. В. Челпанов

Москва 2015 год

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «МАТИ - Российский государственный технологический университет имени К.Э. Циолковского» (МАТИ)

КАФЕДРА
«Моделирование систем и информационные технологии»

У Т В Е Р Ж Д А Ю :
Зав. кафедрой

“_____” _____ 2015 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент Сапин Александр Сергеевич
(Фамилия, Имя, Отчество)

Институт Ступинский филиал МАТИ Специальность 230100.62 «Информатика и вычислительная техника» Группа 14ИВТ-4ДБ-005

1. Тема дипломного проекта (работы):
Разработка компьютерной игры Pacman с элементами искусственного интеллекта на платформе Qt

Утверждена приказом по Университету от 12.05.2015 г. № 605

2. Исходные данные к проекту (работе) (в том числе, указать проектную и технологическую документацию и основную литературу):

1. Алгоритмы. Построение и анализ / Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн / Издательский дом «Вильямс». 2012 г. – 1290 с.
2. UC Berkeley CS188 Intro to AI(online - ресурс) / Dan Klein, Pieter Abbeel / https://www.cs.berkeley.edu/~russell/classes/cs188/f14/
3. Язык программирования C++ / Бьерн Страуструп / Бином. 2011 г. – 1136 с.
4. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес/Питер. 2010 г. – 366 с.

3. Перечень подлежащих разработке вопросов:

3.1. В специальной части проекта:

1. Анализ и изучение существующих алгоритмов искусственого интеллекта.
2. Разработка математической модели.
3. Разработка игры Pacman на основе созданой модели.
4. Реализация выбраных алгоритмов.
5. Разработка уровней для игры.
6. Тестирование приложения.

4. Перечень графического или иллюстративного материала :

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

5. Срок сдачи студентом законченного проекта руководителю «___»_____20__ г.

Дата выдачи задания «22» мая 2015 г.

Руководитель _____ В. В. Лидовский
подпись расшифровка подписи

Подпись студента _____ А. С. Сапин
подпись расшифровка подписи

Примечание:
1. Задание прилагается к законченному проекту и вместе с ним представляется в ГЭК.
2. Студент должен согласовать с руководителем график выполнения работы, отражающий весь период проектирования (с указанием сроков выполнения проекта и трудоемкости отдельных этапов).

Содержание

1 Введение

Системы искусственного интеллекта являются популярной и быстроразвивающейся облатью информатики. Такие системы нашли свое применение в промышленном производстве, военной технике, персональных устройствах, а также в играх. Таким образом, проблема исследования эффективности, а также практического применения различных алгоритмов является актуальной.

Существует множество алгоритмов искусственного интеллекта для игр с полной информацией, таких как шахматы, шашки, крестики-нолики и т.д. Наиболее известным из них является алгоритм минимакс, который, при некоторой оптимизации, показывает хорошие результаты в приведенных вариантах настольных игр. Однако, эффективность минимакса в других типах игр, например, играх с лабиринтом мало изучена. Ещё одним известным примером являются алгоритмы машинного обучения. Они нашли широкое применение в сфере анализа данных, но достаточно редко используются в играх.

Игра Pac-Man является известной игрой в жанре аркады, вышедшая в 1980 году в Японии. Эта игра является ярким представителем игр класса лабиринт с полной информацией. Несмотря на кажущуюся простоту, Pacman является хорошей средой для реализации и тестирования алгоритмов искусственного интеллекта, ввиду сложной структуры лабиринта, наличия очков, а также аппонентов, которых представляют призраки.

2 Постановка задачи

Целью данной выпускной квалификационной работы является разработка компьютерной игры Pacman на платформе Qt и реализации искусственного интеллекта – «бота» для протагониста. Также, необходимо обеспечить возможность выбора конкретного алгоритма, а также безотказность работы.

Задачи:

– анализ и изучение существующих алгоритмов искусственого интеллекта;

– разработка математической модели;

– разработка игры Pacman на основе созданой модели;

– реализация выбраных алгоритмов;

– тестирование;

– документирование.

3 Разработка системы

3.1 Описание системы

Система представляет собой приложение выполненное на платформе Qt версии 5.2. Исходный код программы написан на языке C++ стандарта 2011 года. Программа представляет собой исполняемый файл, файл конфигурации и папку с уровнями. Система разрабатывается с помощью системы контроля версий Git. В качестве репозитория используется GitHub – `https://github.com/alesapin/Pacman`. Для написания кода и компиляции используется IDE QtCreator, версии 3.0.1. При скачивании исходников необходимо выполнить импорт проекта в QtCreator.

3.2 Алгоритмы и модель игры

Для применения и тестирования были выбраны 2 различных алгоритма искусственого интеллекта – минимакс и Q-обучение. Алгоритмы используют совершенно различный подход к решению задачи и выбору действия.

3.2.1 Минимакс

Это алгоритм для минимизации возможных потерь из тех, которые лицу, принимающему решение, нельзя предотвратить при развитии событий по наихудшему для него сценарию. Суть алгоритма заключается в построении дере-

ва возможных ходов, с учетом ходов соперника, и выбор наилучшего исхода.

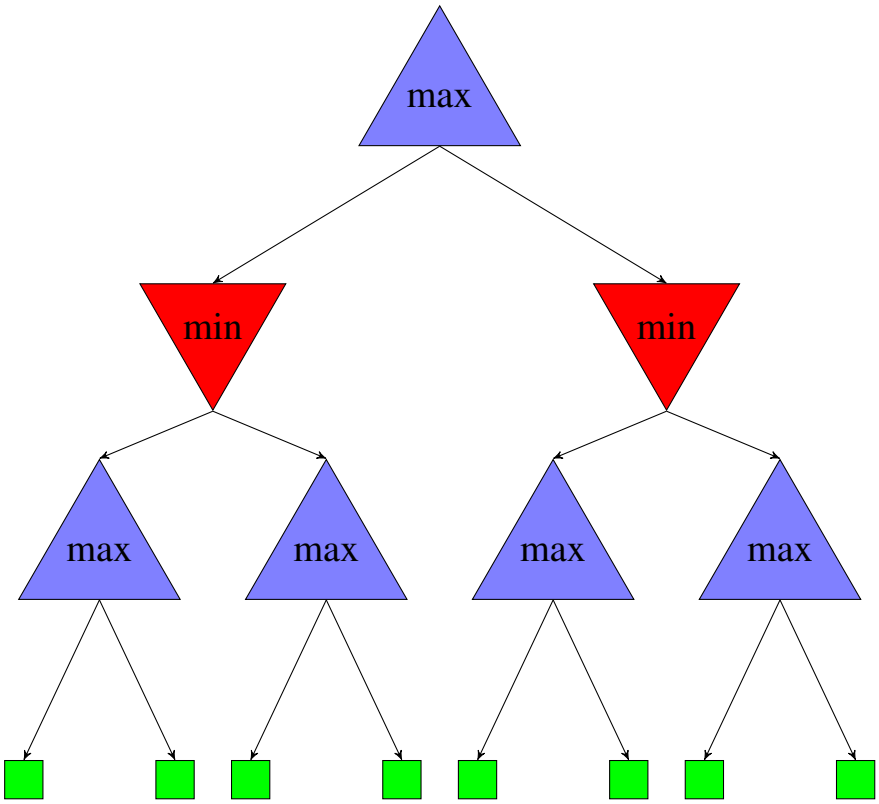


Рисунок 1 — Дерево минимакса

Как видно из рисунка 1, в дереве минимакса происходит поочередная смена между игроком(max) и его противником(min). На каждом уровне игрок выбирает максимум из всех выходов предыдущего уровня, в то время как противник выбирает минимум.

В Pacman'е, как правило присутствует более 1 противника(призрака), поэтому один максимизирующий шаг будет сменяться несколькими, в зависимости от количества призраков, минимизирующими шагами. Псевдокод алгоритма представлен на листинге 1. Из псевдокода и древовидной структуры алгоритма видно, что глубина минимакса может быть очень большой, что сказывается на производительности. Для эффективной реализации минимакса применяется несколько различных оптимизаций:

– альфа-бета отсечение;

– ограничение максимальной глубины.

```
1  function minval(state)
2      v ← −∞ ;
3      foreach successor of state do
4          v ← max(v, value(successor));
5      end
6      return v ;
7  function maxval(state)
8      v ← ∞ ;
9      foreach successor of state do
10         v ← min(v, value(successor));
11     end
12     return v ;
13 function value(state)
14     if state is terminal then
15         return evaluate(state); //Счет состояния
16     end
17     if next agent is MAX then
18         return maxval(state) ;
19     end
20     if next agent is MIN then
21         return minval(state) ;
22     end
23     return v ;
```

Листинг 1: Минимакс

Минимакс с альфа-бета отсечением выполняет эквивалетные оригинальному минимаксу действия, но значительно эффективнее. На рисунке 2 представлено дерево минимакса с альфа-бета отсечением. Из рисунка видно, что некоторые листы (и целые ветви) можно даже не рассматривать (отсекать), т.к. очевидно, что максимизирующему игроку будет выгоднее выбрать уже рассмотренный узел дерева.

Ограничение максимальной глубины является вынужденной необходимо-

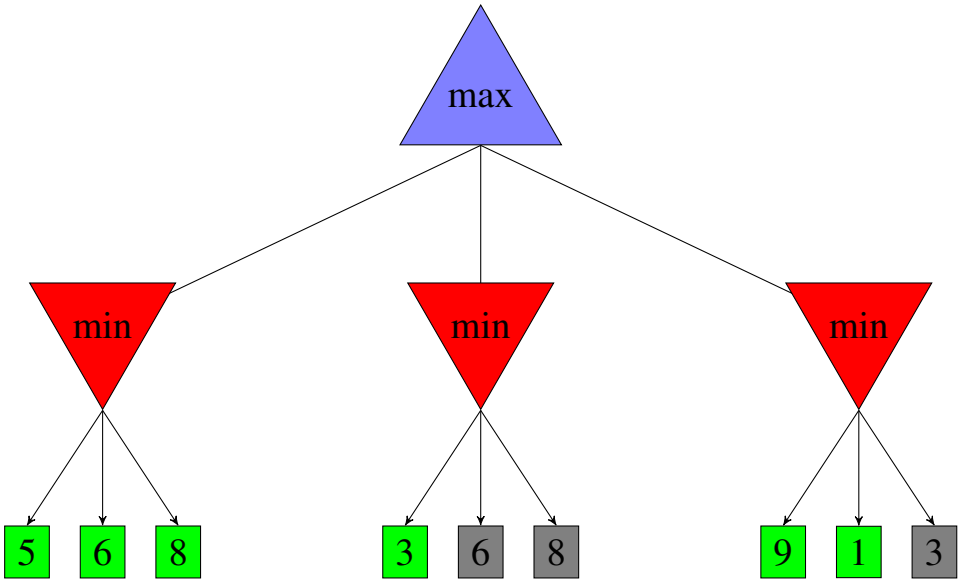


Рисунок 2 — Дерево минимакса с альфа-бета отсечением

стью для минимакса, даже с учетом альфа-бета отсечения, т.к. асимптотическое время работы алгоритма близко к $O(b^d)$, где b - фактор ветвления, а d - высота полученного дерева. Даже такой маленький лабиринт, который представлен на рисунке 3, для расчета первого хода потребует дерева глубиной более 30 уровней, что является очень трудоемкой, для вычисления, задачей. Для решения этой проблемы, в практической реализации минимакс выполня-

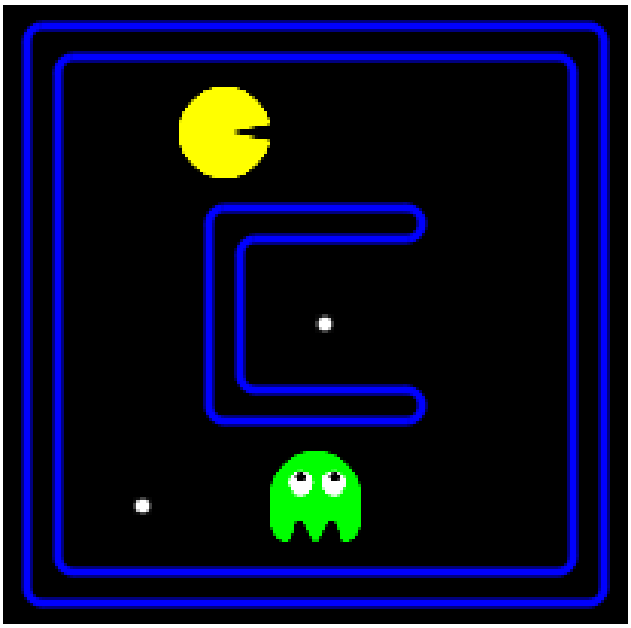


Рисунок 3 — Пример уровня

ется не на полную глубину – до конечных состояний, а на какую-то наперед

заданную.

3.2.2 Вероятностный минимакс

Одним существенным недостатком минимакса является предположение о том, что игра происходит против идеального противника, который всегда минимизирует возможные исходы. Однако, в большинстве реализаций игры Pacman, в том числе и оригинальном Pac-Man, поведение призраков является отчасти случайным. Иными словами, призраки могут действовать, как бы, в ущерб себе. Это, зачастую, ставит «идеального» минимакс-бота в тупик. Поэтому существует модификация минимакса, которая «усредняет» возможные действия противника(-ов) и называется вероятностный минимакс (*англ. expectimax*).

Основное отличие от оригинального минимакса – замена минимизирующих узлов, усредняющими, которые возвращают сумму возможных исходов, умноженных на вероятность этих исходов. Из дерева, представленного на ри-

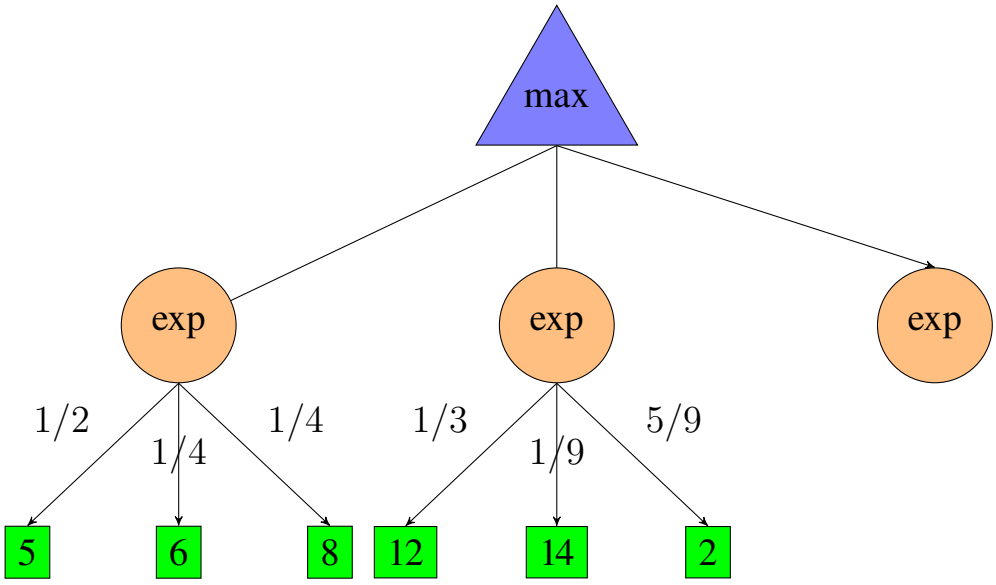


Рисунок 4 — Дерево вероятностного минимакса

сунке 4, видно, что каждый, из возможных исходов самого левого узла имеет определенную вероятность. Также, нетрудно заметить, что альфа-бета отсечение не приминимо в данной ситуации, т.к. чтобы узнать значение конкретного expect-узла, нам нужно знать значение всех его потомков. Алгоритм, в

целом, похож на минимакс, за исключение процедуры *minval*, которая заменяется *expectval*. Псевдокод этой функции приведен на листинге 2.

```
1 function expectval(state)
2     v ← 0 ;
3     foreach successor of state do
4         p ← probability(successor);
5         v ← v + p · value(successor);
6     end
7     return v ;
```

Листинг 2: Функция expectval

3.2.3 Q-обучение

Относится к классу обучающихся алгоритмов с подкреплением. Используется в агентном подходе. В ходе обучения, агент обучается на основании взаимодействия со средой. При этом виде обучения среда представляется в виде Марковского процесса принятия решения. Рассмотрим пример упрощенного представления различных состояний игры в виде цепи Маркова. На рисунке 5 изображена упрощенная цепь Маркова. Все состояния в которых достигается победа объединены в одно состояние $S3$. Все состояния в которых достигается поражение объединены в одно состояние $S2$. В цепи рассматриваются лишь два из всех возможных путей в представленом лабиринте.

Данная цепь описывается:

– семью возможными состояниями $\{S_0, S_1, ..., S_6\}$;

– начальным состоянием S_0;

– конечными состояниями S_2 и S_3;

– тремя возможными действиями:

- пойти направо – обозначено непрерывной оранжевой стрелкой;
- пойти наверх – обозначено фиолетовой пунктирной стрелкой;

- пойти вниз – обозначено синей точечной стрелкой.

– вероятностной функцией $T(s, a, s')$, обозначена красным цветом, над стрелкой;

– функцией вознаграждения $R(s, a, s')$, обозначена зеленым цветом, под сретлкой;

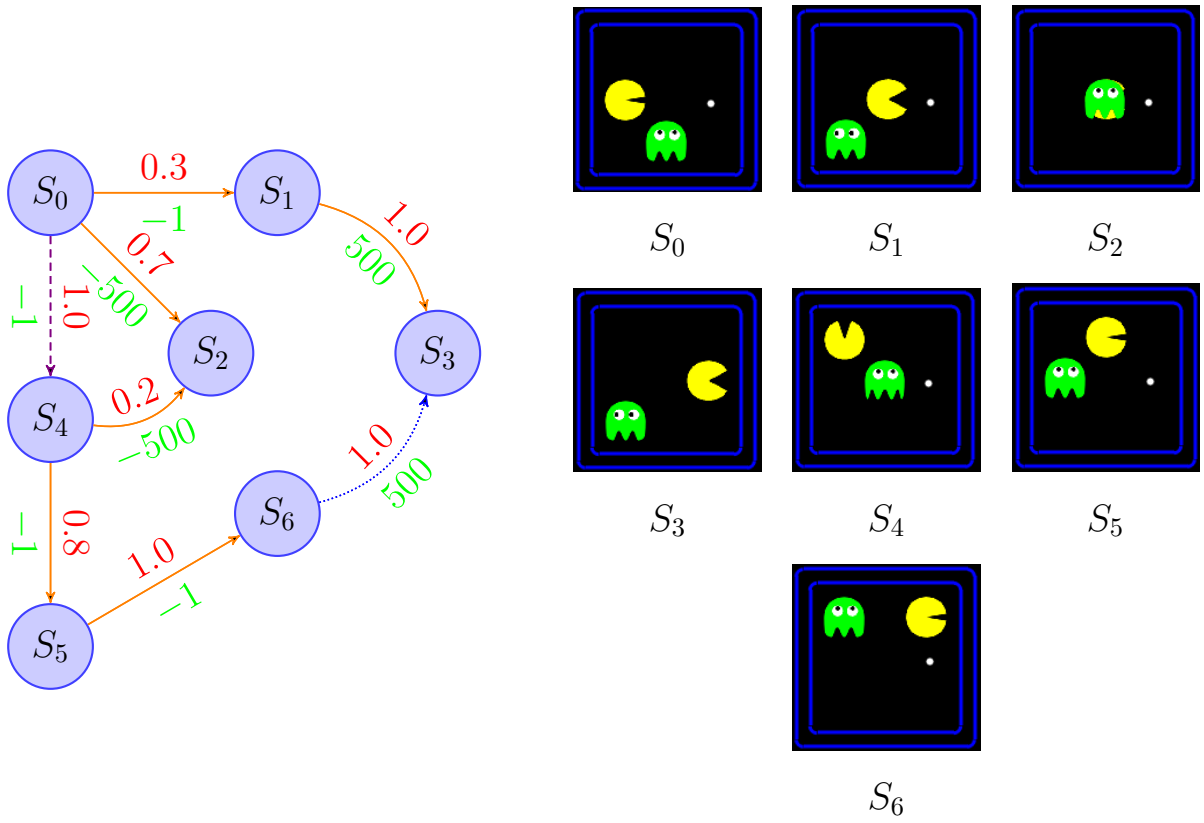


Рисунок 5 — Упрощенная цепь Маркова

Очевидно, что если пойти из состояния S_0 направо – в состояние S_1, то можно достичь победы самым быстрым способом и с наибольшим количеством очков, но вероятность удачного перехода составляет лишь 30%, в остальных случаях Pacman'а настигнет призрак и будет засчитано поражение. С другой стороны, выбрав более длинный путь, через состояние S_4, с большей вероятностью можно достичь победы, но количество очков будет меньшим, по сравнению с первым вариантом. Марковский процесс принятия решения заключается в том, чтобы оценить риски в сложившейся ситуации и принимать те или иные действия.

Для того, чтобы приступить к процессу решения, нужно ввести некоторые обозначения:

– $V^*(s)$ - ожидаемое значение конкретного состояния. Показывает какой результат мы можем получить, в среднем, если будем действовать оптимально из состояния;

– $Q^*(s, a)$ - ожидаемое значение, которое можно будет получить, если из состояния s совершить действие a;

Из определений видно, что для принятия действия в конкретном состоянии нам надо знать значения($V^*(s)$) для каждого из состояний. Для решения этой задачи удобно воспользоваться рекурсивным уравнением Беллмана.

$$V^*(s) = max_a Q^*(s, a) \tag{1}$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s')\right] \tag{2}$$

γ – это уменьшающий фактор, так как $V^*(s')$ это значение, получаемое из последующего состояния, которое наступает в следующий момент времени.

Суть уравнения Беллмана заключается в том, что для каждого состояния s, выбирается максимально возможное(оптимальное) значение $V^*(s)$, среди всех действий, которые мы можем предпринять – $Q^*(s, a)$. Значение, получаемое при совершении действия a, из состояния s – $Q^*(s, a)$ можно расчитать, как среднее от всех возможных состояний s', в которые мы можем попасть, предприняв действие a, учитывая вероятность попадания $T(s, a, s')$ и награду, получаемую в этом состоянии $R(s, a, s')$. Если рассмотреть, уравнение Беллмана приминительно к игре Pacman, то можно обнаружить, что агента, в первую очередь, интересуют действия, которые ему выгодно принимать из данного состояния, а не значения самих состояний. Поэтому нас интересуют Q - значения, а не V -значения, вследствии чего данный алгоритм и называется **Q-обучение**. Тогда уравнение можно записать в виде:

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \cdot max_{a'} Q_k(s', a')\right] \tag{3}$$

Это уравнение представляет собой рекурсивный процесс, выполняя который, будут получены все Q-значения для всех состояний и возможных из них действий. Однако, возникает проблема. В начале игры значения $T(s, a, s')$ и $R(s, a, s')$ не определены, так как не известно, как ведут себя призраки. Их поведение может быть, как полностью случайным, так и достаточно сложным, как в оригинальном Pac-Man. Поэтому, ввиду обозначеной проблемы, эти значения придется узнавать самостоятельно, играя в игру снова и снова(обучаясь), при этом постоянно уточняя Q- значения для состояний и действий, которые совершает Pacman. В результате, изучив лабиринт и поведения призраков, находясь в определенном состоянии, Pacman, будет просто выбирать максимальное действие для этого состояния – $max_a Q(s, a)$. Однако при обновлении Q - значений может возникнуть проблема. Например, исследуя лабирин, Pacman проходил одно состояние множество раз, съедая в нем еду, но однажды в этом состоянии его настиг призрак. Т.к. столкновение с призраком имеет очень большое негативное значение, Q - значение этого состояния станет невыгодным для посещения, несмотря на предудыщие успешные проходы. Для того, чтобы обновление Q-значений происходило плавно, без резких скачков, для новых полученых значений вводится усредняющий параметр α. По другому, его можно назвать коэффициентом уверенности.

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot Q'(s, a) \tag{4}$$

В уравнении (4) $Q(s, a)$ это текущие значение, полученое в результате предыдущих наблюдений. $Q'(s, a)$ это новое значение полученное на текущей итерации. Чем больше значение α, тем быстрее проходит обучение и тем менее адекватна оценка принимаемых действий. В случае маленького значения α процесс обучения больше опирается на предыдущие результаты.

На практике, зачастую, оказывается невозможным реализация Q-обучения в чистом виде, т.к. количество пар (s, a), в некоторых играх, может быть очень велико. К таким играм относиться и Pacman. Поэтому, на практике, используется алгоритм приближенного Q-обучения. Основное отличие от классического алгоритма в том, что пары (s, a) представляются не в явном виде,

а в виде набора «свойств».

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_n f_n(s,a) \quad (5)$$

В формуле (5), $f_i(s,a)$ - функция извлекающая i-ое «свойство» из состояния, w_i - вес (важность) этого свойства. Теперь, мы не можем обновлять Q - значения непосредственно, в процессе обучения. Теперь обучение будет происходить за счет обновления значений весов, для различных свойств.

$$difference = (R(s,a,s') + \gamma max_{a'} Q(s',a')) - Q(s,a) \quad (6)$$

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a) \quad (7)$$

Из формул (6) и (7) видно, что весовые коэффициенты обновляются точно также, как до этого обновлялись Q - значения.

3.2.4 Модель игры

Для реализации выбранных алгоритмов, была разработана модель представляющая игровое поле. Карта представляется матрицей ячеек, каждая из которых может содержать различный объект или сразу несколько – Pacman'а, призраков, еду, капсули, пустоту,стену. Карта является статической и служит основой для состояний. Состояния являются динамическим представлением карты, каждое состояние хранит в себе карту в определенном моменте времени, а также количество очков, соответствующее карте. Состояние может генерировать потомка, в соответствии с принятым действием. Игра происходит, как последовательная смена состояний. Данная модель является хорошим представлнием игры как для минимакса и его вариаций, так и для Q-обучения.

3.3 Структура исходных кодов

Программа написана языке C++ с использованием средств платформы Qt. Исходные коды программы разбиты на несколько каталогов, в которых находятся классы, в соответствии с исполняемыми функциями. В каталоге *engine* находятся классы, отвечающие за общую механику игры. В каталоге *agents*

находятся классы, представляющие собой агентов – ботов, реализующие различные алгоритмы. В каталоге *ui* находятся классы, отвечающие за графическое отображение игры. В каталоге *menu* находятся классы представляющие различные меню. Структура каталогов представлена на рисунке 6.

3.3.1 Структура движка игры (каталог engine)

Каталог *engine* разбит на две логические части - это заголовочные файлы с расширением *.h* и файлы с кодом, с расширением *.cpp*.

Класс *Layout*

Базовым классом, который читает данные непосредственно из файла, является класс *Layout*. Он представляет собой статическую карту. Стены и еда представленны как двумерные булевские векторы. Если в соответствующей ячейки присутствует стена или еда, то в соответствующем массиве значение этой ячейки истино, в противном случае ложно. Положения капсулей и агентов(Pacman'а и призраков) представлены в виде одномерных векторов содержащих точки с их положением. Взаимодействие с классом осуществляется с помощью getter-ов.

Класс *Configuration*

Классом представляющим положение агента, а также позволяющим совершать перемещение, является класс *Configuration*.

Как видно из листинга 3 в этом классе определеяются возможные направления движения, а также метод *generateSuccessor*,который меняет положение position соответственно с переданным вектором.

Класс *Actions*

Классом, отвечающим за корректность совершаемых действий является класс *Actions*. Он содержит в себе исключительно статические методы. Наиболее важным методом этого класса является метод *getPossibleActions* представленный на листинге 4. Метод принимает положение агента и структуру

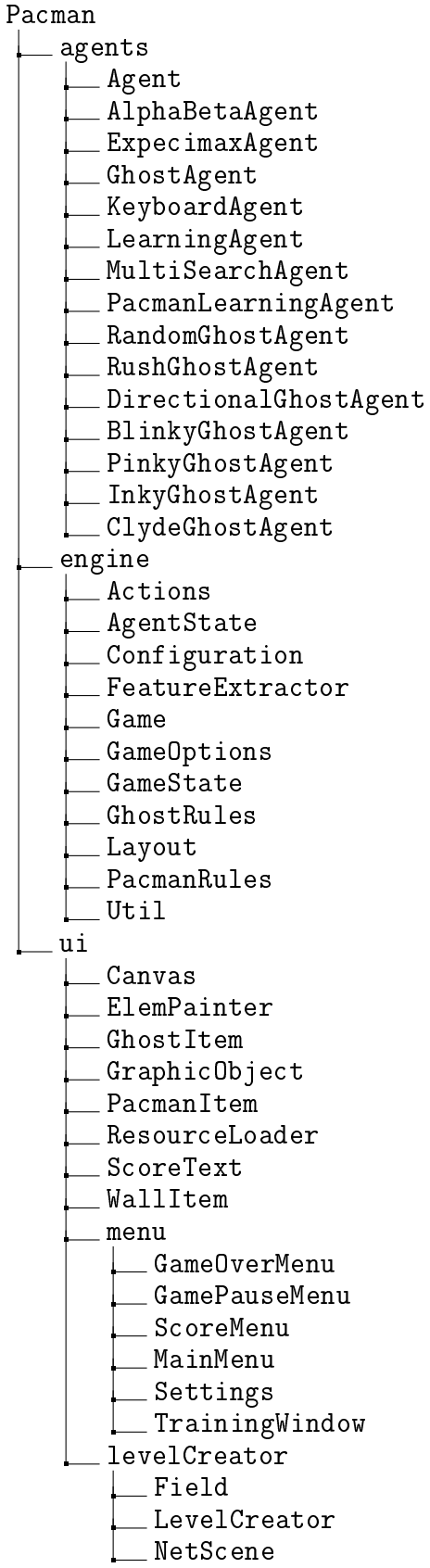


Рисунок 6 — Дерево каталогов с файлами

```
1  enum Direction{
2      NORTH,SOUTH,EAST,WEST,STOP,NOACTION
3  };
4  class Configuration {
5  public:
6    /*...*/
7      Configuration generateSuccessor(QPointF vectorTo);
8  private:
9      Direction direction;
10     QPointF position;
11 };
```

Листинг 3 — Заголовочный файл класс *Configuration*

лабиринта в виде булевского двумерного массива *walls*. Перебирая все существующие направления метод выбирает те, которые не приведут к столкновению со стеной или выходу за пределы лабиринта и возвращает их список. Направление STOP доступно всегда.

Классы *PacmanRules* и *GhostRules*

Правилы игры агентов задаются классами ***PacmanRules*** и ***GhostRules***. также, как и класс ***Actions***, они содержат исключительно статические методы. Наиболее важным методом является метод *applyActions*, который присутствует в обоих классах. Код метода представлен на листинге 5.

Первым делом, метод проверяет переданное направление на доступность. Если действие является доступным, то изменяется конфигурация соответствующего призрака, и устанавливается в состояние. Таким образом и происходит совершение действий в игре.

Класс *GameState*

Состояние в игре представляется классом ***GameState***. Он хранит всю необходимую информацию о текущем состоянии игры, а также позволяет генерировать последующие состояния. Наиболее важным методом класса является метод *generateSuccessor*. На основании переданого номера агента и направле-

```
1  std::vector<Direction> Actions::getPossibleActions(Configuration&
       config, const std::vector<std::vector<bool> >& walls) {
2      std::vector<Direction> result;
3      QPointF pos = config.getPosition();
4      double x = pos.x();
5      double y = pos.y();
6
7      int intx = (int)(x+0.5);
8      int inty = (int)(y+0.5);
9      if (std::abs(x-intx)+std::abs(y-inty) > TOLERANCE){
10         result.push_back(config.getDirection());
11         return result;
12     }
13     for (auto iter = directions.begin();iter != directions.end()
           ;++iter){
14         Direction currentDirection = iter->first;
15         if(currentDirection == STOP) continue;
16         QPointF currentVector = iter->second;
17         double dx = currentVector.x();
18         double dy = currentVector.y();
19         double nextX = x + dx;
20         double nextY = y + dy;
21         double indX = (int)(nextX+0.5);
22         double indY = (int)(nextY+0.5);
23
24         if (nextX < 0 || nextX >= walls.size()) continue;
25         if(nextY < 0 || nextY >= walls[0].size()) continue;
26         if (!walls[indX][indY]) result.push_back(currentDirection
               );
27     }
28     result.push_back(STOP);
29     return result;
30 }
```

Листинг 4 — Метод *getPossibleActions* класса *Actions*

```
1  void GhostRules::applyAction(GameState &state, Direction dir, int
       ghostIndex) {
2      std::vector<Direction> legal = GhostRules::getLegalActions(
           state,ghostIndex);
3      if(std::find(legal.begin(),legal.end(),dir)==legal.end()){
4          qDebug() << "Illegal ghost Direction";
5          throw 2;
6      }
7      AgentState ghostState = state.getAgentState(ghostIndex);
8      double speed = GhostRules::GHOST_SPEED;
9      if (ghostState.getScarryTimer()){
10         speed /= 2.0;
11     }
12     QPointF vect = Actions::directionToVector(dir,speed);
13
14     ghostState.setConfiguration(ghostState.getConfiguration().
           generateSuccessor(vect));
15     state.setAgentState(ghostIndex,ghostState);
16 }
```

Листинг 5 — Метод *applyActions* класса *GhostRules*

ния движения он создает последующее состояние с помощью правил.Код метода приведен на листинге 7. Метод *generatePacmanSuccessor*, является его «фасадом» исключительно для Pacman'а.

Класс *AgentState*

AgentState представляет состояния агента и является более высокоуровневой оберткой над классом ***Configuration***. Помимо конфигурации, класс определяет является ли агент Pacman'ом и время, в течении которого агент будет «испуганым»(исключительно для призраков). Также позволяет сравнивать состояния агентов.

```
1  class GameState {
2  public:
3    GameState(const Layout* lay);
4    GameState(const GameState&);
5    /* ... */
6    std::vector<Direction> getLegalActions(int agentNum) const;
7      GameState* generateSuccessor(int agentIndex, Direction dir);
8      std::vector<Direction> getLegalPacmanActions() const;
9      GameState* generatePacmanSuccessor(Direction dir);
10     /* Getters and Setters */
11     /* ... */
12 private:
13   const Layout* layout;
14     std::vector< std::vector<bool> > food;
15     std::vector< AgentState> agentStates;
16     std::vector<QPointF> capsules;
17     std::vector<QPointF> eaten;
18     bool win;
19     bool lose;
20     int score;
21     QPointF eatenFood;
22     QPointF eatenCapsule;
23 };
```

Листинг 6 — Класс ***GameState***

Класс *GameOptions*

Начальное состояние игры: лабиринт, типы агентов, характеристики агентов задаются конфигурационным файлом. Структурой, представляющей этот файл в программе, является ***GameOptions***. Он представлен на листинге 8. Помимо данных, это структура содержит 2 статических метода – *createDefaultCfg* и *parseFromFile*, позволяющих создать стандартный конфигурационный файл и создать структуру из существующего конфигурационного файла.

```
1  GameState* GameState::generateSuccessor(int agentIndex, Direction
       dir){
2      if(isWin() || isLose()) qDebug() << "Terminal state,lose/win"
           ;
3      GameState* state=new GameState(*this);
4      if(agentIndex == 0){
5          PacmanRules::applyAction(*state,dir);
6      }else{
7          GhostRules::applyAction(*state,dir,agentIndex);
8      }
9      if(agentIndex == 0){
10         state->addScore(-1);
11     }else{
12         state->setAgentState(agentIndex,GhostRules::
               decrementTimer(state->getAgentState(agentIndex)));
13     }
14     GhostRules::checkDeath(*state,agentIndex);
15     return state;
16 }
```

Листинг 7 — Метод *generateSuccessor* класса ***GameState***

Класс *Game*

Главным классом, отвечающим за процесс игры является класс ***Game***. Класс имеет приватный конструктор, поэтому не может быть инстанциирован на прямую. Создание игры происходит с помощью статического метода *parseOptions*, который получает на вход опции игры. Основным методом класса является метод *step*, который совершает один временной шаг в процессе игры и возвращает полученное состояние. Класс хранит в себе счетчик текущего агента, метод *step* инкрементирует этот счетчик, таким образом определяется, кто ходит в этот момент времени.

Также активно используется метод *restartGame*, который позволяет перезапустить игру с уже загруженными настройками, что очень удобно, например, для обучающегося агента. Для предоставления этой возможности класс хранит стартовое состояние игры *startState*. Отдельно для тренировки обуча-

```
1  struct GameOptions {
2      QString pacmanAgent;
3      QString ghostAgent;
4      QString layoutPath;
5      int numIters;
6      double alpha;
7      double epsilon;
8      double gamma;
9      int minimaxDepth;
10     int cellSize;
11     QGraphicsScene* scene;
12     static  void createDefaultCfg(QFile&);
13     static GameOptions *parseFromFile(QFile&);
14 };
```

Листинг 8 — Структура ***GameOptions***

ющегося агента используется метод *trainAgentStep*, а для отслеживания процесса обучения методы-getter'ы *getTotalIters* и *getCurrentIter*. В классе хранится вектор всех агентов, а также определено несколько переменных для агента представляющего обучающегося и клавиатурного Pacman'а, так как они не полностью соответствуют интерфейсу ***Agent***. Класс представлен на листинге 9. Метод *step* на листинге 10.

Класс *Util*

Класс ***Util*** является вспомогательным. Он определяет методы общего назначения, такие как *manhattanDistance* расчитывающее манхэттенское расстояние или *tossCoin* реализующий подбрасывание монеты.

Класс *FeatireExtractor*

Класс ***FeatureExtractor***, извлекает «свойства» из состояния и используется обучающимся агентом.

```
1  class Game {
2  public:
3
4      static Game *parseOptions(GameOptions& opts);
5    /* ... */
6      GameState *step();
7      ~Game();
8      void restartGame();
9      int getCurrentIter() const;
10     int getTotalIters() const;
11     int trainStep();
12   private:
13     GameState* currentGameState;
14     GameState* startState;
15     int currentMover;
16     Layout* layout;
17     PacmanLearningAgent* pacman;
18     KeyBoardAgent* keyboard;
19     std::vector<Agent*> agents;
20     Game(std::vector<Agent*> agents,Layout* lay,bool learn);
21 };
```

Листинг 9 — Класс ***Game***

```
1 GameState *Game::step() {
2     Direction action = agents[currentMover]->getAction(*
        currentGameState);
3     GameState* newState = currentGameState->generateSuccessor(
        currentMover,action);
4     if(currentGameState != startState){
5         delete currentGameState;
6     }
7     currentMover = (currentMover+1)%(agents.size());
8     return currentGameState = newState;
9 }
```

Листинг 10 — Метод *step* класса ***Game***

3.3.2 Агенты (каталог agents)

Все классы, которые представляют в игре агентов наследуют единый интерфейс ***Agent***. Благодаря этому обеспечивается полиморфизм. UML-диаграмма классов представлена на рисунке 7.

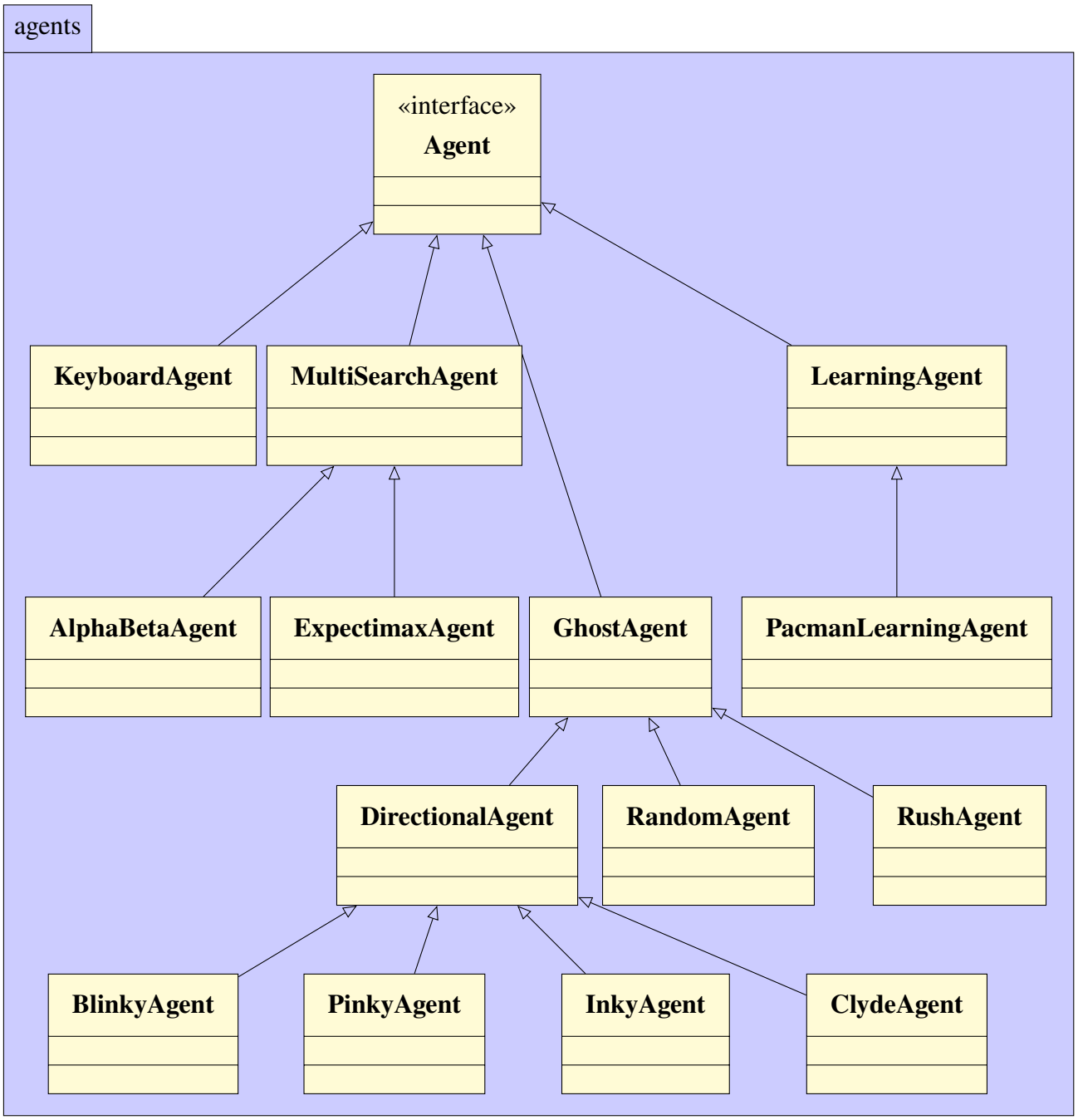


Рисунок 7 — UML-диаграмма классов каталога **agents**

```
1 class Agent {
2 public:
3       virtual Direction getAction(GameState& state) = 0;
4 protected:
5     int index;
6 };
```

Листинг 11 — Интерфейс ***Agent***

Класс Agent

Agent является чисто абстрактным классом и представляет общий интерфейс для всех агентов. Он определяет единственный чисто–виртуальный метод *getAction*, который должны переопределять все наследники класса. также содержит в себе одно поле, представляющее номер агента. Код класса представлен на листинге 11.

Класс *MultiAgentSearch*

MultiAgentSearch представляет собой базовый класс для реализации алгоритмов на основе минимакса. Он добавляет новую, необходимую для реализации алгоритма функцию - *evaluationFunction*, которая возвращает «оценку» состояния, а также поле *depth* необходимое для ограничение алгоритма по глубине. Для расчета оценки функция учитывает расстояние до ближайшей еды, количество оставшейся еды, а также нахождение призраков на расстоянии одного шага. Для расчета расстояния до ближайшей еды используется метод *closestFood* класса ***Util***. Если бы было возможно реализовать минимакс без отсечения, то эта функция бы не понадобилась, т.к. «оценкой» состояния был бы полученный счет. Код функции приведен на листинге 12.

Класс *AlphaBetaAgent*

AlphaBetaAgent является первым классом-наследником класса ***MultiAgentSearch***. Он реализует минимакс с альфа-бета отсечением и ограничением глубины. Для реализации используются 3 дополнительных метода:

minimax,maxVal и *minVal*.Каждый из них исполняет функцию соответствующию псевдокоду представленному на листинге 1. Ограничение по глубине проверяется в методе *minimax*, а инкремент счетчика глубины осуществляется в методах *maxVal* и *minVal*, исключительно, если этот уровень дерева является максимизирующим. Альфа-бета отсечение реализуется введением соответствующих переменных *alpha* и *beta* и преждевременным возвратом из цикла, в случае обнаружения отсечения.

Для возврата из функций нескольких значений используется шаблонный класс tuple, добавленый в стандарте C++ 11. Код переопределенной функций *getAction* приведен на листинге 13, код функций *minimax* на листинге 14. Код функции *maxVal* представлен на листинге 15. Код функции *minVal* практически идентичен *maxVal*, за исключением условий отсечения и изменения значения.

```
1  double MultiAgentSearch::evaluationFunction(GameState &state){
2      double result = 0;
3      QPointF pacmanPosition = state.getPacmanPosition();
4      int totalFood = state.getLayout()->getTotalFood();
5      int size = (state.getLayout()->getWalls()[0].size())*(state.
           getLayout()->getWalls().size());
6      double coef = 1.*totalFood/size;
7
8      double distToFood = Util::closestFood(pacmanPosition, state.
           getFood(), state.getLayout()->getWalls());
9      if(distToFood == -1){
10         result += 1000;
11     }else{
12         result += ((1./ (distToFood+1))*10);
13     }
14     double minDist = std::numeric_limits<double>::infinity();
15     for(int i = 1;i<state.getAgentStates().size();++i){
16         QPointF ghostPosition = state.getAgentPosition(i);
17         minDist = std::min(minDist, Util::manhattanDistance(
               pacmanPosition, ghostPosition));
18     }
19     if(minDist <= 1){
20         result -=5000;
```

```
21      }
22      if(state.getNumFood() == 0){
23          result +=100000;
24      }else{
25          result += ((1./(state.getNumFood()))*100000);
26      }
27      return result;
28  }
```

Листинг 12 — Код функции *evaluationFunction* класса ***MultiAgentSearch***

Класс *ExpectimaxAgent*

ExpectimaxAgent также наследует класс ***MultiSearchAgent***. Он реализует вероятностный минимакс с ограничением по глубине. Код, в целом, совпадает с кодом ***AlphaBetaAgent*** за исключением метода *expectVal*, который заменяет собой метод *minVal*. В цикле выбирается не минимальное возможное значение, а средних среди всех возможных.Код функции соответствует псевдокоду приведенному на листинге 2 .также отсутствует альфа-бета отсечения.

```
1  Direction AlphaBetaAgent::getAction(GameState &state){
2      double alpha = -std::numeric_limits<double>::infinity();
3      double beta = std::numeric_limits<double>::infinity();
4      std::tuple<Direction,double> result = minimax(state,0,0,alpha
           ,beta);
5
6      return std::get<0>(result);
7  }
```

Листинг 13 — Переопределенная функция *getAction* класса ***AlphaBetaAgent***

Класс *LearningAgent*

LearningAgent является базовым классом для обучающихся агентов. Поле *discount*, которое является коэффициентом γ в уравнении Беллмана. Поле *alpha* является усредняющим коэффициентом α из уравнения (4). Поле *epsilon* является коэфициентом определяющим долю случайных действий

```
1  std::tuple<Direction, double> AlphaBetaAgent::minimax(GameState &
       state, int dpth, int agentNum, double alpha, double beta){
2      if(dpth == depth || state.isWin() || state.isLose()){
3          return std::make_tuple (NOACTION, evaluationFunction(state
               ));
4      }
5      if(agentNum == 0){
6          return maxVal(state,dpth,agentNum,alpha,beta);
7      }else{
8          return minVal(state,dpth,agentNum,alpha,beta);
9      }
10 }
```

Листинг 14 — Функция *minimax* класса ***AlphaBetaAgent***

агента. Другими словами, с вероятностью *epsilon* будет выбрано случайное доступное действие вместо опоры на предыдущий опыт. Обучение агента происходит при вызове функции *observationFunction*, которая в свою очередь вызывает функцию *update*, которая должна быть переопределена в наследниках и обновлять $Q(s, a)$ - значения. Количество итераций, доступных для обучения передается в конструктор и храниться в поле *numIters*. Каждая итерация это одна сыгранная игра - эпизод. В начале каждого эпизода вызывается метод *startEpisode*, который сбрасывает последнее действие и состояние агента. В конце каждого эпизода вызывается метод *endEpisode*, который инкрементирует счетчик эпизодов и если максимальное число эпизиодов было достигнуто, сбрасывает значения *alpha* и *epsilon*, для того, чтобы в режиме настоящей игры агент поступал максимально обдумано, опираясь на предыдущий опыт и не совершал случайных действий. также предусмотрены методы-getter'ы для счетчика эпизодов. Код функций, осуществляющих процесс обучения приведен на Листниге 16.

Класс *PacmanLearningAgent*

PacmanLearningAgent является наследником класса ***LearningAgent***. Он реализует методы *update* и *getQValue*, что является необходимым условием.

```
1  std::tuple<Direction, double> AlphaBetaAgent::maxVal(GameState &
       state, int dpth, int agentNum, double alpha, double beta){
2      std::tuple<Direction,double> v =    std::make_tuple(NOACTION
           ,-std::numeric_limits<double>::infinity());
3      std::vector<Direction> availableActions = state.
           getLegalActions(agentNum);
4      int nextAgent = (agentNum+1)%state.getAgentStates().size();
5      if(nextAgent == 0) dpth+=1;
6      for(Direction action:availableActions){
7          GameState* nextState = state.generateSuccessor(agentNum,
               action);
8          std::tuple<Direction,double> innerResult = minimax(*
               nextState,dpth,nextAgent,alpha,beta);
9          delete nextState;
10         if(std::get<1>(innerResult) > std::get<1>(v)){
11             v = std::make_tuple(action,std::get<1>(innerResult));
12         }
13         if(std::get<1>(v) > beta) return v;
14         alpha = std::max(alpha,std::get<1>(v));
15     }
16     return v;
17 }
```

Листинг 15 — Функция *maxVal* класса ***AlphaBetaAgent***

```
1  void LearningAgent::observeOneAction(GameState &state, Direction
       action, GameState &nextState, double deltaReward) {
2      update(state,action,nextState,deltaReward);
3  }
4
5  GameState *LearningAgent::observationFuction(GameState &state) {
6      if(lastAction!=NOACTION){
7          double reward = state.getScore() - lastState.getScore();
8          observeOneAction(lastState,lastAction,state,reward);
9      }
10     return &state;
11 }
```

Листинг 16 — Обучающие функции класса ***LearningAgent***

Обновление Q-значений сведено к итерационному процессу. Изначальная идея, о поиске значений в виде $Q(s, a)$ плохо показывает себя на практике, т.к. даже для маленькой карты, количество Q-значений очень велико. Поэтому, в реализации класса, используется поиск Q-значений не для каждого из возможных состояний, а для «свойств» состояний - приближенное Q-обучение. «Свойства» можно получить с помощью класса ***FeatureExtractor***. Каждая пара (s, a) раскладывается в набор «свойств», а именно:

– bias – базовое свойство, всегда равно 1;

– close-ghost – в зависимости от наличия призрака на расстоянии шага равно 1 или 0;

– eat-food – равно 1, если совершив действие a, Pacman окажется в ячейки с едой;

– closest-food – расстояние до ближайшей еды.

Весовые коэффеценты каждого из свойств храняться в ассоциативном массиве в виде пар имя_свойства–вес.

Таким образом для принятие решения, о том, куда двигаться из состояния s, мы раскладываем каждую возможную пару (s, a) в набор «свойств», считаем для каждой пары Q-значение по формуле (5) и выбираем ту пару(то действие), Q - значение которого является максимальным. Код функций, реализующих это действие приведен на листинге 18.

Функция производящая процесс обучения представлена на листинге 17.

```
1 void PacmanLearningAgent::update(GameState &state, Direction
    action, GameState &nextState, double reward){
2
3     std::map<std::string,double> feature = FeatureExtractor::
        getFeatures(state,action);
4     std::vector<double> qVals;
5     for(Direction act:nextState.getLegalPacmanActions()){
6         qVals.push_back(getQValue(nextState,act));
7     }
8     if(qVals.empty()){
```

```
9           qVals.push_back(0);
10      }
11      double maxVal = *std::max_element(qVals.begin(),qVals.end());
12      difference =  reward+ discount*maxVal - getQValue(state,
            action);
13      for(auto iter=feature.begin();iter!=feature.end();++iter){
14          weights[iter->first] += alpha*difference*iter->second;
15  }
16
17  double PacmanLearningAgent::getQValue(GameState &state, Direction
        action)
18  {
19      double qVal = 0;
20      std::map<std::string,double> feature = FeatureExtractor::
            getFeatures(state,action);
21      for(auto iter=feature.begin();iter!=feature.end();++iter){
22          qVal += weights[iter->first]*iter->second;
23      }
24      return qVal;
25  }
```

Листинг 17 — Функции для обучения в классе ***PacmanLearningAgent***

```
1   Direction PacmanLearningAgent::getAction(GameState &state)
2   {
3       GameState& currentState = *(this->observationFuction(state))
            ;
4       std::vector<Direction> legalActions = currentState.
            getLegalPacmanActions();
5       Direction action = NOACTION;
6       if(Util::tossCoin(epsilon)){
7           int index = rand() % legalActions.size();
8           action =  legalActions[index];
9       }else{
10          action =  computeActionFromQValues(currentState);
11      }
12      doAction(currentState,action);
13      return action;
14  }
15  Direction PacmanLearningAgent::computeActionFromQValues(GameState
```

```
    &state)
16 {
17     std::vector<Direction> legalActions = state.
           getLegalPacmanActions();
18     if(legalActions.empty()){
19         return NOACTION;
20     }
21     Direction maxAction = NOACTION;
22     double maxVal = -std::numeric_limits<double>::infinity();
23     for(Direction act:legalActions){
24         double currentVal = getQValue(state,act);
25         if(currentVal > maxVal){
26             maxVal = currentVal;
27             maxAction = act;
28         }
29     }
30     return maxAction;
31 }
```

Листинг 18 — Функции для выбора действия в классе ***PacmanLearningAgent***

Класс *GhostAgent*

GhostAgent наследует класс ***Agent*** и представляет собой базовый класс для всех агентов призраков. Он реализует метод *getAction*, а также объявляет чисто виртуальный метод *getDistribution*, который предназначен для расчета значений возможных действий из состояния. Метод *getAction* делает выбор действия на основании этого распределения. Его код представлен на листинге 19.

```
1 Direction GhostAgent::getAction(GameState &state)
2 {
3     QPointF agentPosition = state.getAgentPosition(index);
4     std::map<Direction,double> dist = getDistribution(state);
5     std::vector<std::pair<Direction,double> >listRepr;
6     std::copy(dist.begin(),dist.end(),std::back_inserter(listRepr
          ));
7
```

```
8       std::sort(listRepr.begin(),listRepr.end(),PairComparator());
9
10      double choice = Util::randDouble();
11      int i =0;
12      double total = listRepr[i].second;
13      while(total < choice){
14          total += listRepr[++i].second;
15      }
16      return listRepr[i].first;
17  }
```

Листинг 19 — Функция для выбора действия в классе ***GhostAgent***

Класс *RandomGhostAgent*

RandomGhostAgent является простейшим агентом для призраков. Он всегда выбирает случайное действие. Значения всех доступных для него действий полностью эквивалентны.

Класс *RushGhostAgent*

RushGhostAgent является умным агентом-призраком. Его поведение также отчасти случайно, но в большинстве случаев он предпочитает двигаться на Pacman'а и, если напуган, избегать Pacman'а. Код функции его функции *getDistribution* приведен на листинге 20. Хаотичность действий задается переменными *probScaried* и *probAtack*.

```
1   std::map<Direction, double> RushGhostAgent::getDistribution(
        GameState &state)
2   {
3       AgentState ghostState = state.getAgentState(index);
4       std::vector<Direction> legalActions = state.getLegalActions(
            index);
5       QPointF position = ghostState.getPosition();
6       double speed = state.isScared(index) ? 0.5:1;
7
8       std::vector<QPointF> actionsVector;
9       for (Direction action : legalActions) {
```

```
10          actionsVector.push_back(Actions::directionToVector(action
                ,speed));
11      }
12      std::vector<QPointF> newPositions;
13      for(QPointF vect:actionsVector){
14          newPositions.push_back(QPointF(position.x()+vect.x(),
                position.y()+vect.y()));
15      }
16      QPointF pacmanPosition = state.getPacmanPosition();
17
18      std::vector<double> distancesToPacman;
19       for(QPointF newPos:newPositions){
20           distancesToPacman.push_back(Util::manhattanDistance(
                pacmanPosition,newPos));
21       }
22       double bestScore = 0;
23       double bestProb = 0;
24       if(state.isScared(index)){
25          bestScore = *std::max_element(distancesToPacman.begin(),
                distancesToPacman.end());
26          bestProb = probScaried;
27       } else {
28          bestScore = *std::min_element(distancesToPacman.begin(),
                distancesToPacman.end());
29          bestProb = probAtack;
30       }
31       std::vector<Direction> bestActions;
32       for (int i = 0; i < distancesToPacman.size() ; ++i) {
33           if(distancesToPacman[i] == bestScore){
34               bestActions.push_back(legalActions[i]);
35           }
36       }
37       std::map<Direction,double> result;
38       for (Direction act : legalActions) {
39           result[act] = 0.0;
40       }
41       for (Direction best : bestActions) {
42          result[best] = bestProb / bestActions.size();
43       }
```

```
44        for(Direction legal:legalActions){
45            result[legal] += (1-bestProb) / legalActions.size();
46        }
47        Util::normalize(result);
48        return result;
49    }
```

Листинг 20 — Функция для расчета значений действий ***RushGhostAgent***

Класс *DirectionalGhostAgent*

DirectionalGhostAgent является ещё более умным агентом призраков, чем ***RushGhostAgent***. Находясь в любой позиции он выстраивает кратчайший путь до Pacman'а, используется алгоритм *BFS* и двигается соответственно этому пути. Код функции,находящейся в классе ***Util***, определяющей действие приведен на листинге 21.

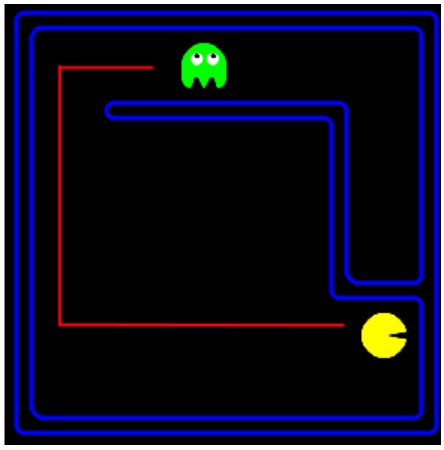
```
1  Direction Util::ghostWayToPoint(int ghostNumber, QPointF point,
       const GameState &state)
2  {
3      std::deque<std::tuple<Configuration,Direction>> fringe;
4      std::vector<Direction> legal = state.getLegalActions(
           ghostNumber);
5      fringe.push_back(std::make_tuple(state.getAgentState(
           ghostNumber).getConfiguration(),legal[0]));
6      std::set<QPointF,PointComparator> expanded;
7      if(state.getLayout()->getWalls()[point.x()][point.y()]){
8          point = emptyNearWall(point,state.getLayout()->getWalls()
               );
9      }
10     bool firstStep = true;
11     while(!fringe.empty()){
12         std::tuple<Configuration,Direction> current = fringe.
               front();
13         fringe.pop_front();
14         Configuration conf = std::get<0>(current);
15         QPointF pos = conf.getPosition();
16         Direction startDirection = std::get<1>(current);
17         if(expanded.find(pos)!=expanded.end()){
```

```
18                continue;
19            }
20            expanded.insert(pos);
21            if(pos == point){
22                return startDirection;
23            }
24
25            std::vector<Direction> legal = GhostRules::
                  getLegalActions(std::get<0>(current),state.getLayout()
                  ->getWalls());
26
27            if(firstStep){
28                for(Direction dir:legal){
29                    Configuration succ = conf.generateSuccessor(dir);
30                    fringe.push_back(std::make_tuple(succ,dir));
31                }
32                firstStep = false;
33            }else{
34                for(Direction dir:legal){
35                    Configuration succ = conf.generateSuccessor(dir);
36                    fringe.push_back(std::make_tuple(succ,
                          startDirection));
37                }
38            }
39        }
40
41        return legal[rand()%legal.size()];
42    }
```

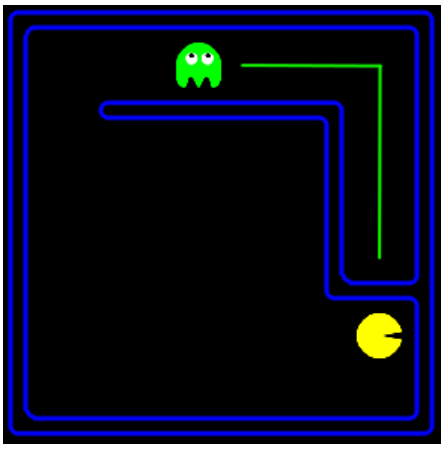
Листинг 21 — Функция для выбора действия при движении к точке класса ***Util***

Классы *BlinkyGhostAgent,PinkyGhostAgent,InkyGhostAgent,ClydeGhostAgent*

Все четыре класса являются наследниками ***DirectionalGhostAgent*** и переопределяют его метод *countTarget*, который определяет, в какую точку будет двигаться призрак. Они представляют собой призраков из классического варианта Pac-Man.



(а) ***DirectionalGhostAgent***



(б) ***RushGhostAgent***

Рисунок 8 — Различие между ***DirectionalGhostAgent*** и ***RushGhostAgent***

BlinkyGhostAgent - призрак преследователь, он полностью совпадает с направленым призраком. Его целью всегда является Pacman.

PinkyGhostAgent - призрак, который является помощником ***BlinkyGhostAgent***, он всегда стремится оказаться на 4 клетки впереди Pacman'а, если Pacman движется вверх, то на 4 клетки впереди и слева от Pacman'а.

InkyGhostAgent - самый опасный из 4-х призраков. Своей целью он выбирает конец отрезка, начало которого находится в точке, где расположен ***BlinkyGhostAgent***, а середина в точке, где находится Pacman. Предсказать его действия очень сложно.

ClydeGhostAgent - призрак со странным поведением. Если он находится дальше чем на 9 клеток от Pacman'а, то действует как ***BlinkyGhostAgent***, в противном случае следует в правый нижни угол карты.

Так же, все призраки из 4-ки переодически переходят в режим блуждания, в котором каждый из них следует в свой угол лабиринта. Время блуждания и время преследования определяются игрой, изначально время преследования - 20 секунд, время блуждания - 5 секунд, с каждой сменой поведения, время блуждания уменьшается на 1 секунду, а время преследования увеличивается.

Класс *KeyboardAgent*

KeyboardAgent является агентом, считывающим действия с клавиатуры. Ввиду особенностей обработки событий в Qt, считывание действий вынесено

в другой класс.

3.3.3 Графика (каталог ui)

Классы, представляющие графическое отображение, используют Qt Graphics View Framework. UML-диаграмма классов представлена на рисунке 9. Все

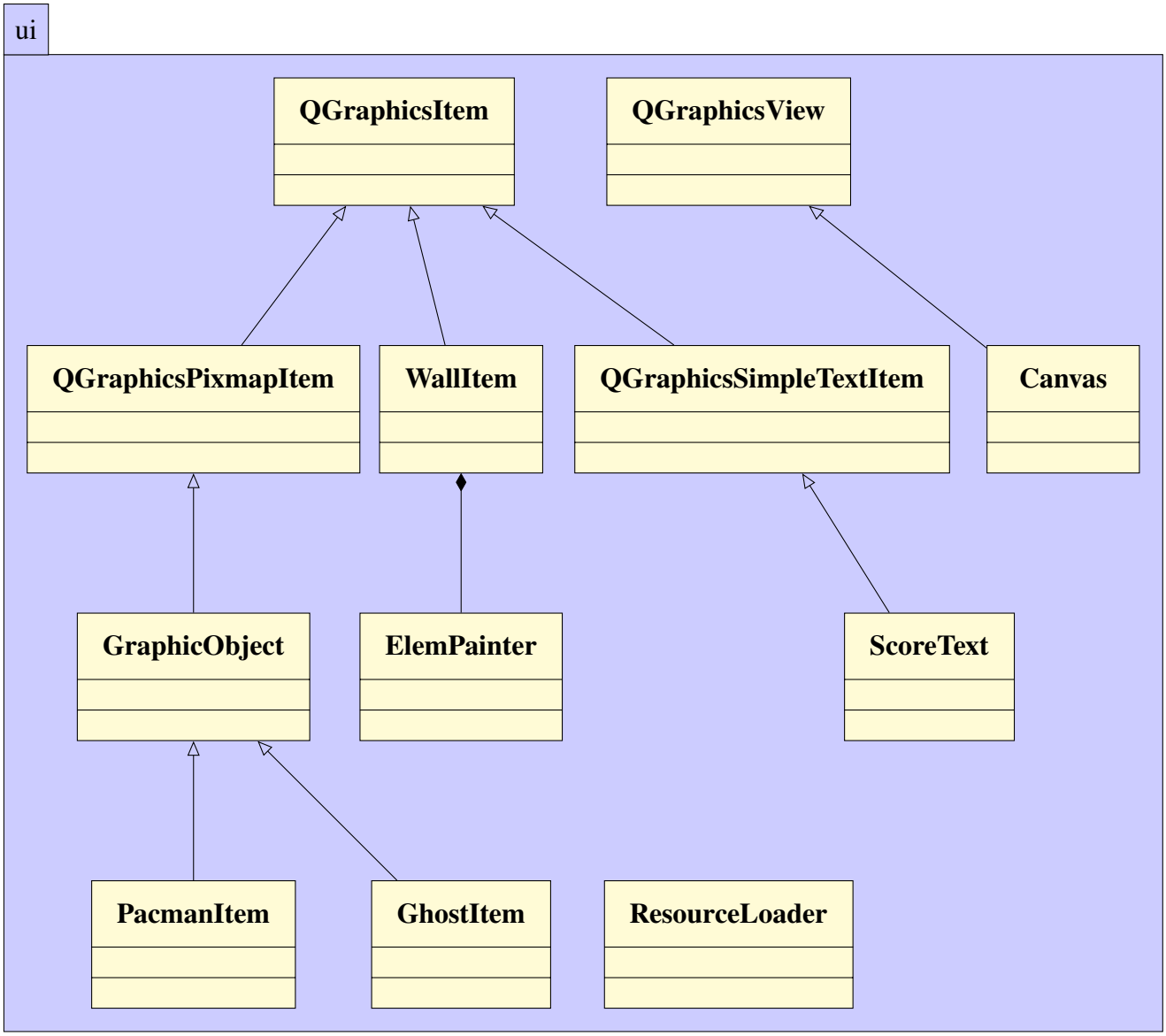


Рисунок 9 — UML-диаграмма классов в пакете ui

классы с приставкой **Q** являются стандартными Qt-классами.

Классы *WallItem* и *ElemPainter*

WallItem является наследником класса ***QGraphicsItem***, который является базовым для объектов, добавляемых на сцену. ***WallItem*** переопределяет метод *paint*, который отвечает за отрисовку на сцене.

На основании переданного двумерного вектора класс отрисовывает на сцене лабиринт, состоящий из множества плиток. Все возможные варианты плиток описаны в перечислении приведенном на листинге 22.

```
1 enum ElemType{
2       FIRST,SECOND,THIRD,FOURTH,LEFT,RIGHT,UP,DOWN,FULL,LEFT_CROSS,
          RIGHT_CROSS,UP_CROSS,DOWN_CROSS,FULL_CROSS,HORIZ,VERT,NONE
3 };
```

Листинг 22 — Перечисление представляющее типы плиток

Для выяснения, что именно надо отрисовать в данной плитке используется метод *checkCell*, который анализирует все соседние плитки и решает, что должно быть отрисовано. Плиточная структура показана на рисунке 10.

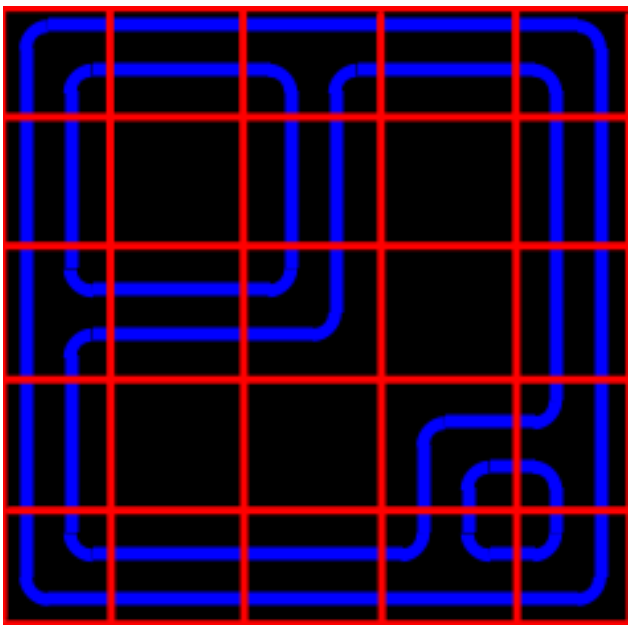


Рисунок 10 — Плиточная структура лабиринта

Отрисовкой в ячейках занимается класс ***ElemPainter***. Фигуры, в каждой ячейке, состоят из нескольких прямых линий и дуг. Пример отрисовки одной фигуры показан на рисунке 11. Красными квадратами обозначены дуги,

а зелеными прямоугольниками прямые, таким образом фигура в этой ячейки состоит из 2-х дуг и 4-х линий. Стоит отметить, что ***WallItem*** использует

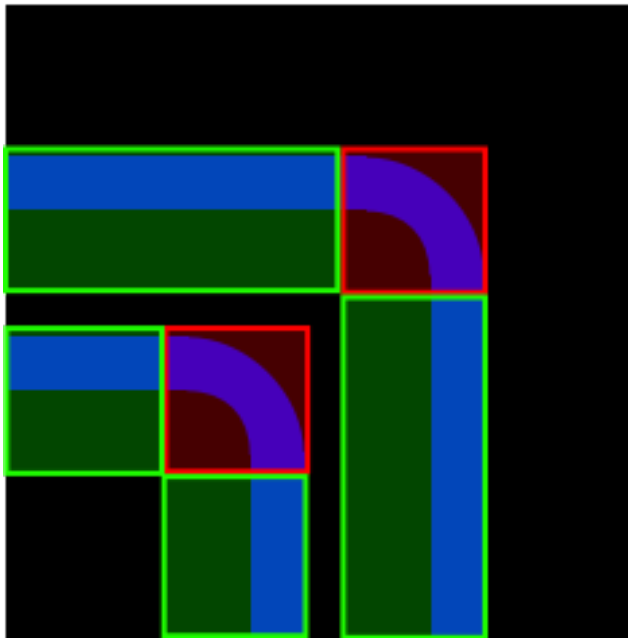


Рисунок 11 — Пример одной плитки

кэширование - *ItemCoordinateCache*, для того, чтобы не происходила перерисовка статического лабиринта при обновлении сцены.

Класс *GraphicObject*

GraphicObject является базовым классом для всех движущихся объектов в игре, а точнее для Pacman'а и призраков. Он является классом-наследником ***QGraphicsPixmapItem***, который предназначен для отображения объектов с наложением текстур. Движение объекта происходит с помощью таймера, который по времени вызывает метод *moveOneStep*. Этот метод смещает объект на один не большой шаг по направлению к цели. Для начала движения объекта используется метод *moveToPoint*. Код конструктора класса приведен на листинге 23.

Класс *PacmanItem*

PacmanItem наследует класс ***GraphicObject*** и является графическим представлением Pacman'а. Движение и повороты обеспечиваются переопределе-

```
1 GraphicObject::GraphicObject(QPointF start, int cs,int stepTime):
2     startPosition(start),
3     cellSize(cs),
4     stepTime(stepTime) {
5     QPointF first(startPosition.y()*cellSize,startPosition.x()*
          cellSize);
6     setPos(first);
7     timer = new QTimer();
8     connect(timer,SIGNAL(timeout()),this,SLOT(moveOneStep()));
9 }
```

Листинг 23 — Конструктор класса ***GraphicObject***

нием родительского метода *moveToPoint*. Код метода приведен на листинге 24. ***PacmanItem*** использует одну текстуру, приведенную на рисунке 12.

Рисунок 12 — Текстура Pacman'а

Класс *GhostItem*

GhostItem также наследует класс ***GraphicObject*** и является графическим представлением призраков. Он переопределяет метод *moveToPoint*, в котором происходит смена текстуры призрака в зависимости от направления движения или испуга. Пример текстур одного призрака приведен на рисунке 13.

Класс *Canvas*

Canvas является самым крупном классом в приложении. Является наследником класса ***QGraphicsScene***. Основной задачей класса является корректное отображение игрового процесса на сцене. Класс содержит поле *game* – объект класса ***Game***, с помощью таймера вызывает его метод *step* и произ-

```
1  void PacmanItem::moveToPoint(QPointF moveTo,Direction dir){
2      switch(dir){
3          case NORTH:
4              setRotation(270);
5              break;
6          case SOUTH:
7              setRotation(90);
8              break;
9          case EAST:
10             setRotation(0);
11             break;
12         case WEST:
13             setRotation(180);
14             break;
15         case STOP:
16             break;
17     }
18     if(currentTarget != moveTo){
19         setPos(currentTarget);
20         timer->stop();
21         currentTarget = moveTo;
22     }
23     setPixmap(texture.copy(currentFrame*cellSize,0,cellSize,
           cellSize));
24     currentFrame=(currentFrame+1)%5;
25     timer->start(stepTime);
26 }
```

Листинг 24 — Код функции *moveToPoint* класса ***PacmanItem***

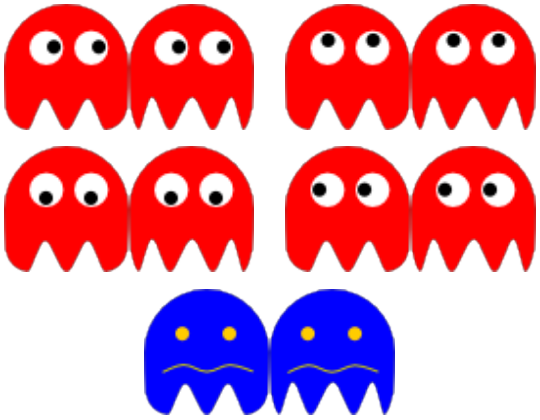


Рисунок 13 — Пример текстуры призрака

водит отрисовку игровых состояний в методе *drawStep*. Код метода приведен на листинге 25.

Pacman,призраки и лабиринт представляются с помощью ранее определенных классов. Еда и капсули представляют собой объекты класса Qt-класса ***QGraphicsEllipseItem*** и хранятся в ассоциативных массивах *capsuleMap* и *foodMap*, с помощью которых их удобно удалять со сцены.

На сцене размещаются кнопки *restart* и *pause*, которые обеспечивают возможность остановить или перезапустить игру. Для реализации этих кнопок используется класс ***QPushButton***. С помощью метода *connect* к кнопкам привязываются действия. Стили кнопок задаются в специально формате *.qss*, храняться в отдельном в файле, внутри ресурсов приложения. Пример стиля приведен на листинге 26.

```
QPushButton {
    background-color: black;
    color: white;
    border-style: outset;
    border-width: 3px;
    border-radius: 10px;
    border-color: blue;
}
```

Листинг 26 — Стиль кнопок в приложении

также на сцене находится счет, который является объектом класса ***ScoreText*** и отрисовывается с помощью бесплатного пиксильного шрифта Munro.

```
1  void Canvas::drawState(GameState *state){
2      if(state->isWin() || state->isLose()){
3          timer->stop();
4          gameOver = true;
5      }
6      QPointF pos = state->getAgentPosition(0);
7
8      pacman->moveToPoint(QPointF(pos.y()*cellSize,pos.x()*cellSize
           ),state->getAgentState(0).getDireciton());
9      for(int i = 0;i<ghosts.size();++i){
10         GhostItem* currentGhost = ghosts[i];
11         QPointF ghostPos = state->getAgentPosition(i+1);
12         currentGhost->moveToPoint(QPointF(ghostPos.y()*cellSize,
               ghostPos.x()*cellSize),state->getAgentState(i+1).
               getDireciton());
13         if (state->isScared(i+1)) currentGhost->scarryMode();
14         else currentGhost->normalMode();
15     }
16     QPointF eatenFood = state->getEatenFood();
17     if(foodMap.find(eatenFood)!=foodMap.end()){
18         scene()->removeItem(foodMap[eatenFood]);
19         delete foodMap[eatenFood];
20         foodMap.erase(eatenFood);
21     }
22     QPointF eatenCapsule = state->getEatenCapsule();
23     if(capsuleMap.find(eatenCapsule)!=capsuleMap.end()){
24         scene()->removeItem(capsuleMap[eatenCapsule]);
25         delete capsuleMap[eatenCapsule];
26         capsuleMap.erase(eatenCapsule);
27     }
28     scoreText->updateScore(state->getScore());
29 }
```

Листинг 25 — Код функции *drawState* класса ***Canvas***

```
1  void Canvas::keyPressEvent(QKeyEvent *event) {
2      if (event->key() == Qt::Key_Escape) {
3          this->setDisabled(true);
4          GamePauseMenu* p = new GamePauseMenu(this);
5          pause = false;
6          this->pauseGame();
7          p->show();
8      }else{
9        game->keyBoardEvent(event);
10     }
11 }
```

Листинг 27 — Перехват клавиатурных событий

Ввиду того, что в Qt, только один объект может отслеживать события клавиатуры, класс ***Canvas*** переопределяет метод *ketPressEvent*, в котором передает сообщения клавиатурному агенту, если он существует и обрабатывает собственные события. Код Метод Приведен на листинге 27.

Пример отрисованной сцены со всеми объектами приведен на рисунке 14.

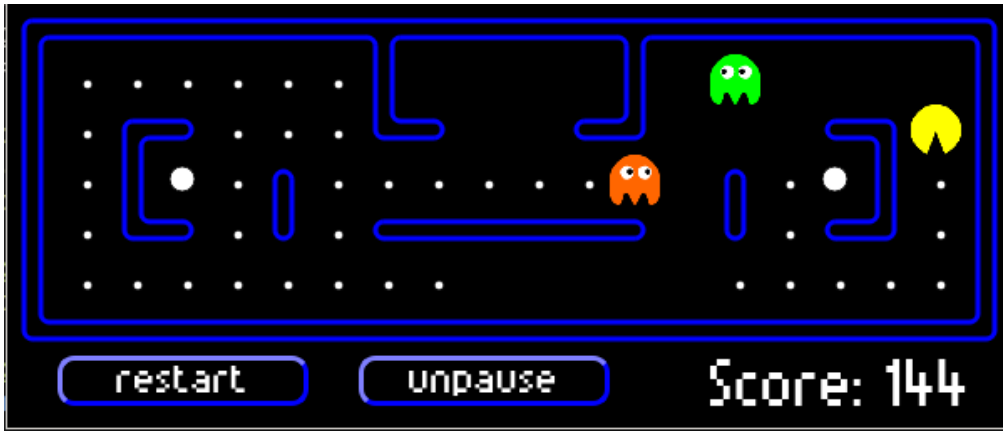


Рисунок 14 — Пример отрисованной сцены

Класс *ScoreText*

ScoreText является наследником класса ***QGraphicsSimpleTextItem***. Реализует отображение текста для счета. Добавляет один новый метод – *updateScore*.

Класс *ResourceLoader*

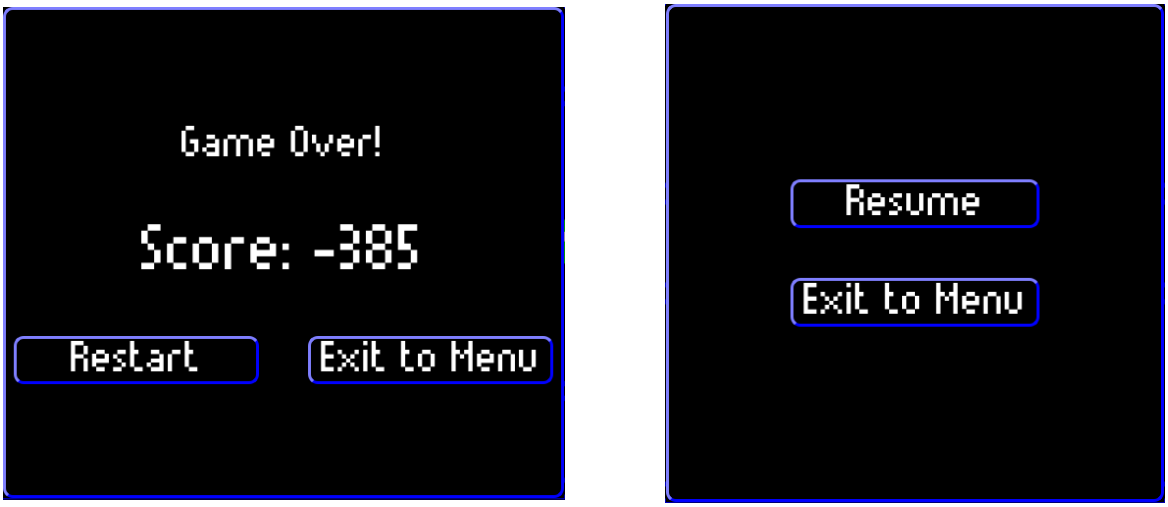
ResourceLoader используется для загрузки некоторых ресурсов, например, стилей. также содержит некоторые константы необходимые для различных классов каталога *ui*.

3.3.4 Меню (каталог ui/menu)

Все классы каталога являются наследниками класса ***QWidget*** и реализуют меню использующиеся в процессе игры.

Классы *GameOverMenu* и *GamePauseMenu*

Оба класса представляют собой небольшие окна, которые всплывают в соответственно при окончании игры и при паузе. С помощью специальной опции у окон убрана рамка. Оба окна представлены на рисунке 15.



(а) Меню конца игры

(б) Меню паузы

Рисунок 15 — Внутриигровые меню

Класс *MainMenu*

MainMenu является главным меню игры, которое появляется при запуске приложения. Позволяет начать игру, перейти в меню настроек, перейти в таблицу очков,запустить редактор уровней и выйти из игры. Представлено на рисунке 16.

Рисунок 16 — Главное меню

Класс *TrainingWindow*

TrainingWindow позволяет визуализировать процесс обучения. Это меню содержит полосу прогресса, которая является объектом QProgressBar и кнопку «Stop Learning», которая позволяет остановить процесс обучения в любой момент и начать игру с текущими результатами. Это меню приведено на рисунке 17.

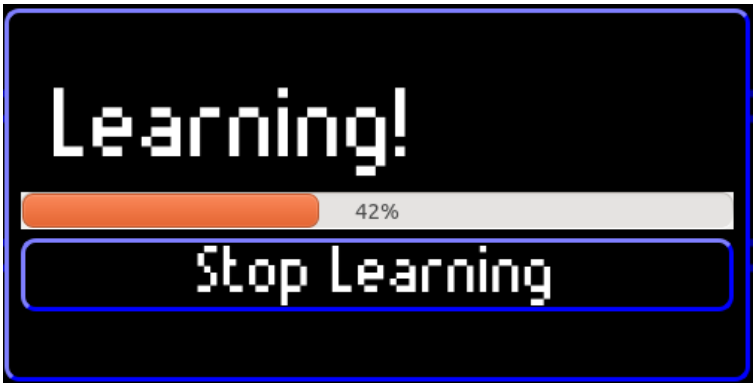


Рисунок 17 — Меню загрузки обучения

Класс *Settings*

Settings представляет собой наиболее сложный класс из всех классов каталога **menu**. Именно считывает и изменяет конфигурационный файл – *config.cfg* в соответствии с действиями пользователя. Если конфигурационного файла не окажется в папке, класс создаст стандартный вариант этого файла. В зависимости от выбранного агента, различные поля в меню становятся доступными. Меню позволяет выбрать:

– агента игрока;

– агента призраков;

– размер ячеек;

– уровень;

– параметры обучающегося агента: α, ϵ,γ и число итераций;

– глубину минимакса;

– скорость игры;

– наличие звука.

Это меню приведено на рисунке 18.

Класс *ScoreMenu*

ScoreMenu является классом представляющим таблицу очков. Для каждого уровня предоставляется топ-10 результатов каждого из агентов Pacman'а. Для хранения результатов используется файл *score.lst* в корневом каталоге приложения. Внешний вид приведен на рисунке 19.

3.3.5 Редактор уровней (каталог ui/levelCreator)

Классы данного каталога реализуют возможность создания новых и редактирования существующих уровней. Макет редактируемого уровня пред-

Рисунок 18 — Меню настроек

ставляется схематически. Минимальный размер уровня 5×5, максимальный 30×30. UML-диаграмма каталога приведена на рисунке 20.

Класс *NetScene*

NetScene является наследником класса ***QGraphicsScene***. К обычному поведению этого класса он добавляет отрисовывание сетки на заднем фоне, для удобства редактирования уровня.

Класс *Field*

Field класс является наследником класса ***QGraphicsView***. Класс реализует основное поле редактора уровней, в котором и просходит "рисование"уровня. В зависимости от выбраного типа "кисти"могут быть отрисованы стены, Pacman, призраки, еда, капсули, пустое пространство. Стены представляют собой синий квадрат. При создании класса сразу создается обрамляющая рамка, которая необходима по условиям игры. Удалить рамку нельзя. Pacman изображается с помощью желтого круга. Так как в игре возможен только один протагонист, при попытке создания нескольких Pacman'ов, лишние будут удалятся. При попытке стереть всех Pacman'ов, новый Pacman появится в случайной ячейке. Призраки представляются маленьким красным квадратом. Максимальное количество призраков - 4, при попытке создать больше призраков,

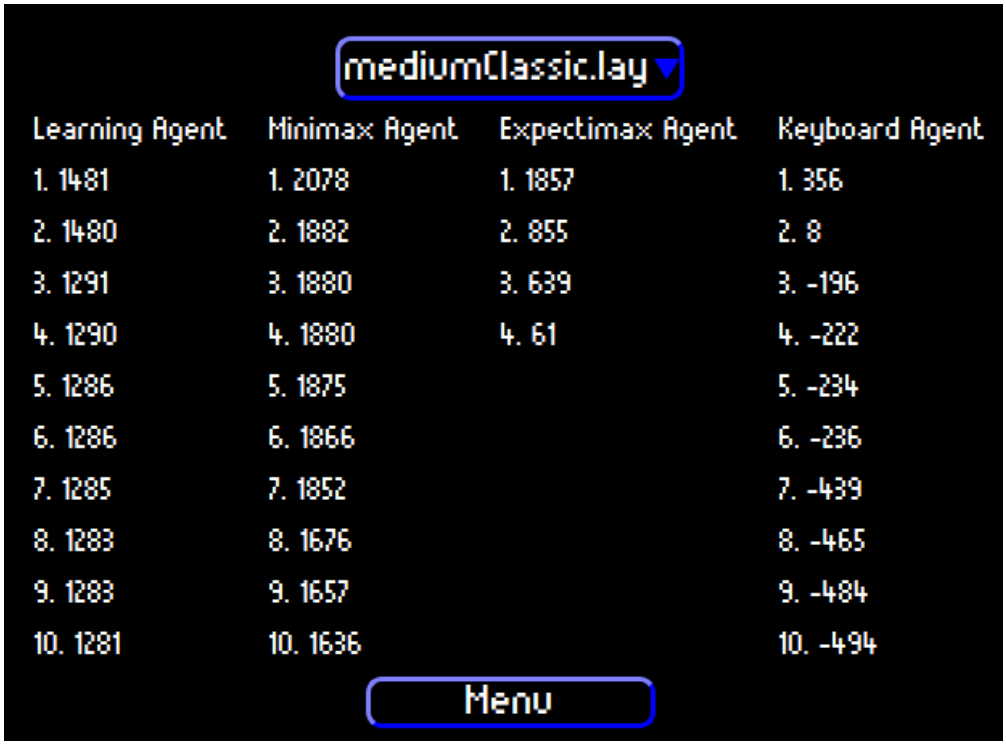


Рисунок 19 — Таблица очков

старые будут удалятся. Еда представляется белой точкой. Капсули представляются маленьким белым кругом. Для обозначения свободного пространства используются черные квадраты разделенные сеткой. Для сцены используется класс ***NetScene***. Класс поддерживает возможность изменения размера с сохранением отрисованных частей, если это возможно. Поле редактора уровней представлено на рисунке 21.

Класс *LevelCreator*

LevelCreator является наследником класс ***QWidget***, содержит в себе поле - ***Field*** и интерфейс управления для него. Управление представлено нижней панелью выбора "кисти"и боковой панелью управления размером поля и сохранения.

Нижняя панель представлена группой из 6-ти кнопок, каждая из которых представляет возможный вариант "кисти". На каждой кнопке установлен сигнал, который при нажатии изменяет тип "кисти"в ***Field*** на соответствующий. Внешний вид нижней панели представлен на рисунке 22. Боковая панель представляет собой вертикальную группу кнопок, слайдеров и поля ввода.

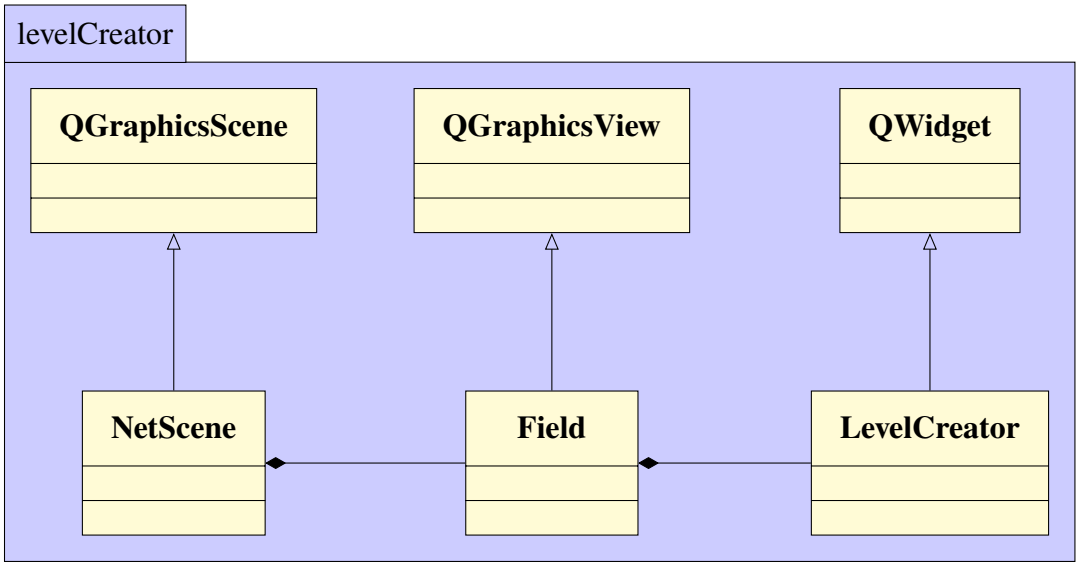


Рисунок 20 — UML-диаграмма классов в пакете ui/levelCreator

С помощью слайдеров *Width*,*Height* и кнопки *Resize* можно изменять размер поля. *W* и *H* показывают текущие размеры поля. Кнопка *Open Layout* позволяет загрузить уже существующий уровень (*.lay* - файл) для редактирования. Кнопка *Save* и поле ввода позволяют сохранить созданный уровень с указанным именем в папку *layouts* приложения. Кнопка *Menu* позволяет выйти в главное меню. Внешний вид боковой панели представлен на рисунке 23.

3.4 Структура приложения

Приложение представляет из себя каталог, который включает в себя исполняемый файл, а также файл *config.cfg* и папку *layouts*, которая содержит в себе файлы уровней – *.lay*. Дерево необходимых файлов приведено на рисунке 24.

3.4.1 Файлы уровней

Для задания уровней используется специальный формат файлов *.lay*. В этом формате приняты некоторые обозначения:

– **%** - стены;

– **.** - еда;

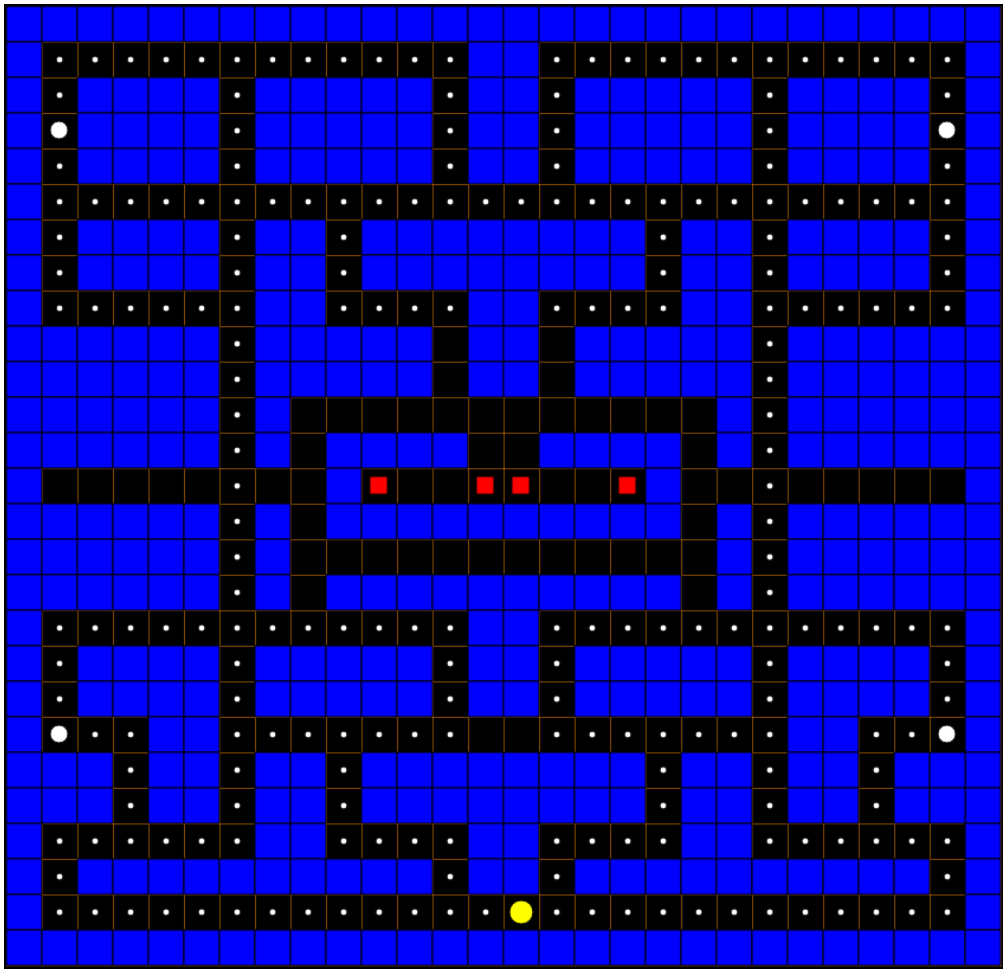


Рисунок 21 — Поле редактора уровней

Рисунок 22 — Нижняя панель управления

– **o** - капсуль;

– **P** - Pacman;

– **G** - призрак;

– ␣- пустая игровая ячейка.

Все уровни, которые будут загружены в игру должны быть закрытыми, т.е. в каждом *.lay* - файле должна находиться рамка из символов **%**. Наличие обозначеных символов за пределами рамки недопустимо. Наличие в файле

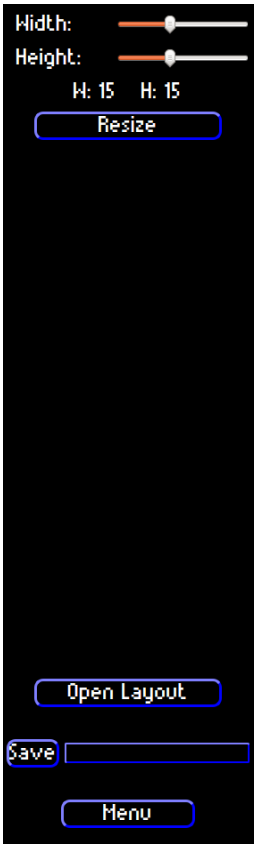


Рисунок 23 — Боковая панель управления

любых других символов, кроме обозначеных, недопустимо. Пример корректного файла приведен на листинге 28.

3.4.2 Файл конфигурации

Для сохранения настроек, выбраных пользователем используется специальный файл конфигуации *config.cfg*. Файл получается с помощью генерации из меню настроек. Его наличие в каталоге не обязательно. В случае отсутствия файла будет сгенерирован файл по-умолчанию. В структуре файла используются следующие обозначения:

– layoutPath - путь к выбранному уровню;

– pacmanAgent - тип агента для Pacman'а, возможны следующие варианты: LEARNING, EXPECTIMAX, MINIMAX, KEYBOARD;

– alpha - значение параметра α, указывается только в случае pacmanAgent=LEARNING;

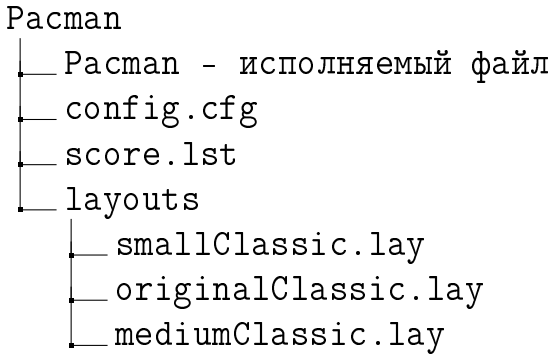


Рисунок 24 — Дерево каталогов приложения с файлами

– epsilon - значение параметра ϵ, указывается только в случае pacmanAgent=LEARNING;

– gamma - значение параметра γ, указывается только в случае pacmanAgent=LEARNING;

– maxDepth - значение максимальной глубины минимакса, указывается только в случае pacmanAgent=MINIMAX или pacmanAgent=EXPECTIMAX;

– ghostAgent - тип агента для призраков, возможны следующие варианты: RANDOM, RUSH,DIRECTIONAL,ORIGINAL;

– cellSize - размер ячеек на поле;

В избежании ошибок рекомендуется не редактировать файл вручную, а пользоваться интерфейсом меню настроек.

3.4.3 Файл с таблицей очков

Для сохранения результатов игр используется специальный файл *score.lst*, который обновляется в процессе игры. Общая структура похожа на *config.cfg*. В файле присутствуют записи только об уровнях, на которых завершилась хотя бы 1 игра.

3.4.4 Исполняемый файл

Исполняемый файл представляет собой бинарный файл, имеющий формат, в зависимости от операционной системы:

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %     %       P         %     %
3  %     %                 %     %
4  % %%%                   %%%% %
5  % %    . .          . . . . .  % %
6  % %   . . . .       . . . . .  % %
7  %  %. .     . .       . . .    % %
8  %   %. . . . . .       . . .    % %
9  %  %. .     . .       . . .    % %
10 %  %. .     . .       . . . . .  % %
11 %  %. .     . .       . . . . .  % %
12 % %                           % %
13 % %%%                   %%%% %
14 %     %   %%%%%%%   %     %
15 %     %   %%%%%%%   %     %
16 %%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Листинг 28 — Пример корректного *.lay* файла

– для Windows – PE;

– для Linux – ELF;

– для Mac OS – Mach-O.

Все необходимые ресурсы, кроме обозначеных выше, находятся внутри исполняемого файла. Это достигается специальными средствами платформы Qt. При разработке все текстуры, стили, шрифты и изображения добавляются в специальный *.qrc* файл, который после компиляции, с помощью ***rcc***, оказывается вложен в исполняемый файл.

4 Заключение

При разработке выпускной квалификационной работы, была поставлена задача разработать компьютерную игру Pacman с использованием алгоритмов искусственного интеллекта. Для достижения этой цели были изучены и отобраны соответствующие алгоритмы, а также разработана математическая

модель. Средствами платформы Qt и языка C++, было разработано приложение соответствующее всем заявленым требованиям.

Таким образом цель выпускной квалификационной работы была достигнута. В ходе работы решены следующие задачи:

– изучены и проанализированы алгоритмы искусственного интеллекта;

– разработана математической модели;

– разработана игра Pacman на основе созданой модели;

– реализованы выбранные алгоритмы;

– приложение протестировано.

5 Использованная литература

1. Алгоритмы. Построение и анализ / Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн / Издательский дом «Вильямс». 2012 г. – 1290 с.

2. UC Berkeley CS188 Intro to AI(online - ресурс) / Dan Klein, Pieter Abbeel / `https://www.cs.berkeley.edu/~russell/classes/cs188/f14/`

3. Язык программирования C++ / Б. Страуструп / Бином. 2011 г. – 1136 с.

4. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес / Питер. 2010 г. – 366 с.