Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ ПО КУРСУ
Алгоритмы компьютерной графики

НА ТЕМУ:
Создание мобильной игры, обучающей основам программирования

Студент ИУ9-52Б _____ В. Е. Пичугин
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ И. Э. Вишняков
(Подпись, дата) (И.О.Фамилия)

Москва 2019 г.

Содержание

Введение

В рамках выполнения данной курсовой работы требуется разработать игру под ОС Android, ключевыми составляющими которой будут алгоритмы визуализации игрового поля и интерпретации кода, управляющего движениями игровых объектов.

Реализации этих алгоритмов должны удовлетворять следующим требованиям:

1. Алгоритм визуализации должен корректно обрабатывать JSON-файл с конфигурацией игрового уровня и строить по нему игровое поле;
2. Помимо этого требуется обеспечение плавных анимаций некоторых игровых объектов;
3. Также пользователь должен иметь возможность создавать свои собственные игровые уровни и сохранять их конфигурации в формате JSON;
4. Алгоритм интерпретации должен обеспечивать корректное исполнение действий игрового персонажа в соответствии с написанным пользователем кодом.

1. Обзор предметной области

1.1. Формирование требований к игровому процессу

Для выбора необходимых для разработки игры инструментов и средств, предоставляемых фреймворком Android, были сформированы следующие базовые требования к игровому процессу:

1. Пользователю должны быть доступны 2 режима:
    a. конструктор уровней - для создания собственных игровых полей.
    b. игровой - для прохождения предустановленного или собственного уровня.
2. визуальное представление конструктора уровней должно содержать:
    a. редактор - для того чтобы пользователь мог взаимодействовать с приложением: управлять типом отрисовываемого игрового объекта, его удалением и сохранением конфигурационного файла созданного уровня на своем мобильном телефоне.
    b. игровое поле с возможностью размещения на нем игровых объектов.
3. визуальное представление игрового режима должно содержать:
    a. редактор кода - для того чтобы пользователь мог комфортно создавать необходимую для прохождения уровня программу и запускать интерпретатор для визуализации написанных команд.
    b. игровое поле, на котором расположены игровые объекты, совершающие анимированные действия при интерпретации кода.

1.2. Обзор инструментов Android для создания игры

Базовым компонентом для построения Android-приложения является Activity - компонент, который представляет собой одиночный экран с пользовательским интерфейсом, и с которым пользователи могут взаимодействовать для выполнения каких-либо действий. Для взаимодействия Activity между собой и системой используются объекты специального класса Intent [1].

Однако для разработки более гибкого пользовательского интерфейса используются объекты Fragment (фрагменты), представляющие собой часть Activity со своей разметкой и жизненным циклом, связанным с хост-Activity. Как правило, при написании Android-приложения логика по управлению пользовательским интерфейсом сосредотачивается во фрагментах, а Activity используются лишь для навигации [2, 3].

Пользовательский интерфейс в Android состоит из структурных элементов, называемых виджетами. Они могут выводить текст или графику, взаимодействовать с пользователем или размещать другие виджеты на экране [4]. Каждый виджет является экземпляром класса View или одного из его подклассов. Один из наиболее важных подклассов View - это ViewGroup, являющийся контейнером для дочерних виджетов. Файлы с разметками для компонентов приложения хранятся в формате XML в папке res/layout рабочей директории. Виджеты в файле с разметкой образуют иерархию (см. Рисунок 1). Иерархию View из XML-файла также можно воспроизвести в Java-коде с помощью объекта специального класса LayoutInflater. Графические файлы приложения хранятся в папке res/drawable рабочей директории в адаптированном под разные конфигурации устройств виде [5].

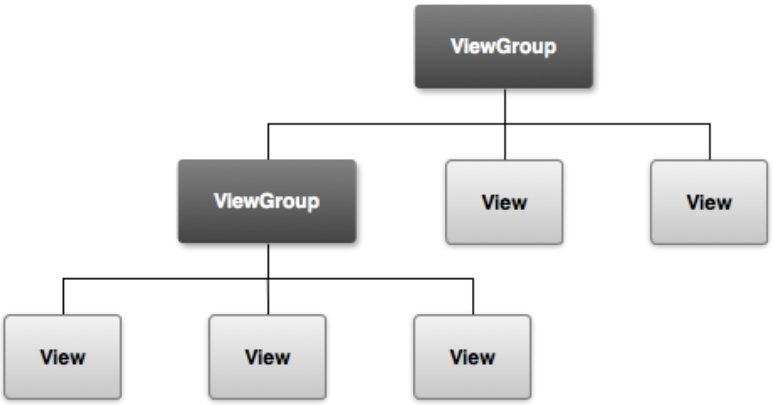

Рисунок 1. Иерархия View.

Для создания игрового поля и отрисовки на нем игровых объектов и анимированных перемещений некоторых из них необходимо создать собственный подкласс View. Для корректного рендеринга игровых объектов следует переопределить метод onDraw(Canvas canvas). Класс Canvas содержит все выполняемые операции графического вывода. Методы, вызываемые для объекта Canvas, определяют, где и что выводится. Чтобы вновь перерисовать созданный подкласс View следует вызвать у него метод invalidate() [4, 6]. Для программного хранения растровых изображений игровых объектов необходимо использовать специальный класс Bitmap.

В процессе формирования требований к игровому процессу обнаружилась необходимость в создании списков внутри приложения - списка доступных уровней, списков строк кода и команд в редакторе. Для этого в Android есть виджет RecyclerView [7, 8], архитектура которого схематично представлена ниже (см. Рисунок 2). Основными его составляющими являются:

1. LayoutManager - отвечает за размещение и измерение элементов, скроллинг и за то, какие элементы больше не нужны;
2. ViewHolder - переиспользуемый элемент списка для повышения производительности, хранящий представление элемента списка и обрабатывающий клики по нему;

3. Adapter - создает View и ViewHolder, заполняет ViewHolder данными и оповещает RecyclerView об изменениях в списке.

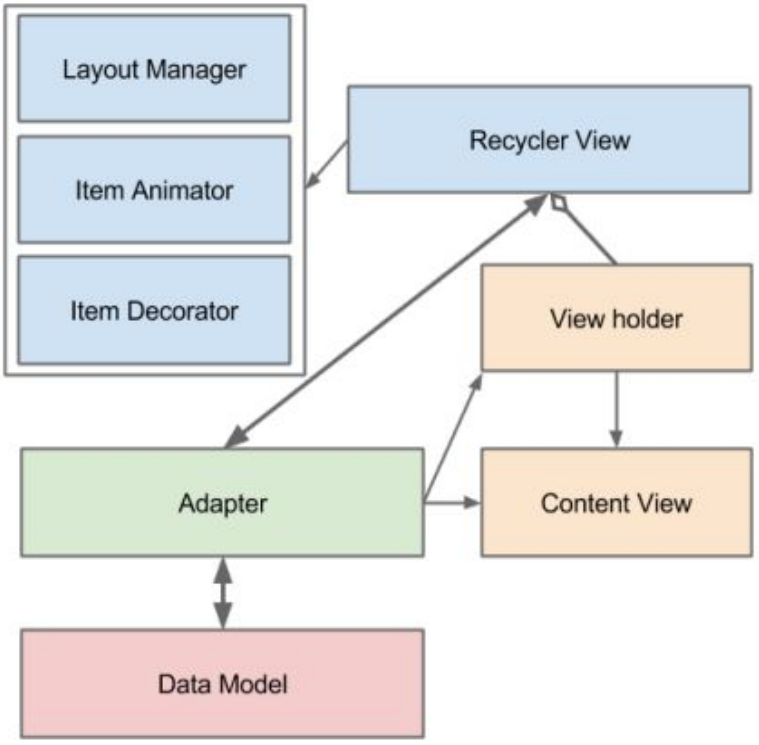

Рисунок 2. Архитектура RecyclerView.

Как было отмечено при формировании требований к алгоритму визуализации игрового поля, приложение должно уметь преобразовывать содержимое конфигурационного Java-объекта в JSON-файл и обратно. Для этой цели можно использовать библиотеку GSON, которая с помощью простых методов toJson() и fromJson() позволяет преобразовать объекты Java в JSON и наоборот [9].

JSON-файлы с конфигурациями предустановленных игровых уровней удобно хранить в папке assets, в которой образуется нечто вроде миниатюрной файловой системы, которая упаковывается в состав приложения [4]. А для хранения конфигурационных файлов для пользовательских уровней удобно использовать Internal Storage - хранилище данных в памяти телефона, где по умолчанию установлено приложение, и доступ к файлам которого имеет только данное приложение [10].

2. Проектирование приложения

2.1. Проектирование классов игрового режима

Для проектирования игры использовался объектно-ориентированный подход с использованием языка программирования Java. Каждая логическая совокупность классов расположена в отдельном пакете.

Начать рассмотрение архитектуры приложения стоит с пакета dungeon, в котором расположены классы приложения, отвечающие за работу игрового поля в режиме игры. Само игровое поле представляет собой двумерную область 8x8 одинаковых квадратов, в каждом из которых находится игровой объект.

UML-диаграмма данных классов представлена на Рисунке 3, а список их описаний представлен ниже:

1. DungeonFragment - фрагмент, отвечающий за часть экрана, в которой расположено игровое поле. Его представление состоит из специально созданного для игры подкласса View - DungeonView. Также фрагмент предоставляет интерфейс для взаимодействия с DungeonView;
2. DungeonView - специально написанный для игры подкласс View, управляющий созданием и отрисовкой игровых объектов, а также реагирующий на игровые события: перемещение персонажа и перезагрузку игры;
3. DungeonPart - абстрактный класс, являющийся базовым для всех игровых объектов. Он содержит базовую позиционную информацию и объект Bitmap для хранения картинки с изображением игровой сущности. Также в нем присутствует стандартный метод отрисовки игрового объекта на DungeonView. Абстрактным же является метод создания Bitmap для изображения в силу того, что у каждой игровой сущности уникальное визуальное представление;
4. Floor - класс, представляющий стену или пол подземелья;

5. Hero - класс, представляющий главного персонажа игры. Помимо стандартной позиционной информации он содержит в специальном объекте PositionPair номера конкретных строки и столбца, где находится персонаж. Также он имеет несколько объектов вспомогательного класса HeroImages для хранения картинок на каждое состояние персонажа во время анимации движения, а также логику для осуществления анимированного перемещения персонажа. Важную роль в этом играет набор колбэков для разных этапов анимации, хранящийся в объекте вспомогательного класса HeroActions;
6. Monster - класс, представляющий врага главного персонажа, на одну клетку с которым попадаться нельзя;
7. Trap - класс, представляющий ловушку на игровом поле - стрелу, летящую в одном из четырех направлений относительно пользователя (слева, сверху, снизу и справа), от которой игровой персонаж должен уклоняться. Данный класс содержит позиционные данные ловушки, информацию об ее типе (она представлена переменной типа TrapType, который является перечислением), а также логику анимации ловушки;
8. Treasure - класс, представляющий сокровище, которое игровой персонаж должен найти. Пользователь побеждает, когда игровой персонаж встает на одну клетку с сокровищем;
9. DungeonConfig - класс, содержащий необходимую для отрисовки игрового поля позиционную информацию об игровых объектах. Именно объекты данного класса сохраняются в виде конфигурационных JSON-файлов и обратно;
10. TrapConfig - вспомогательный класс для хранения конфигурационной информации о ловушках: типе и позиции на игровом поле. Объекты данного класса используются в DungeonConfig;

11. DungeonGenerator - класс, содержащий методы, которые позволяют получить объект DungeonConfig из JSON-строки, а также сгенерировать матрицу стен игрового поля по имеющемуся объекту DungeonConfig;
12. EndgameFragment - подкласс DialogFragment, отвечающий за отображение диалогового окна с возможностью перезагрузить уровень в ответ на возникновение игровых событий, после которых дальнейшее продолжение игры невозможно (победа или поражение пользователя);
13. DialogEventListener - интерфейс, в котором объявлены методы для реакции на важные игровые события - необходимость показать финальное диалоговое окно, сообщение об ошибке, перезагрузить игру.

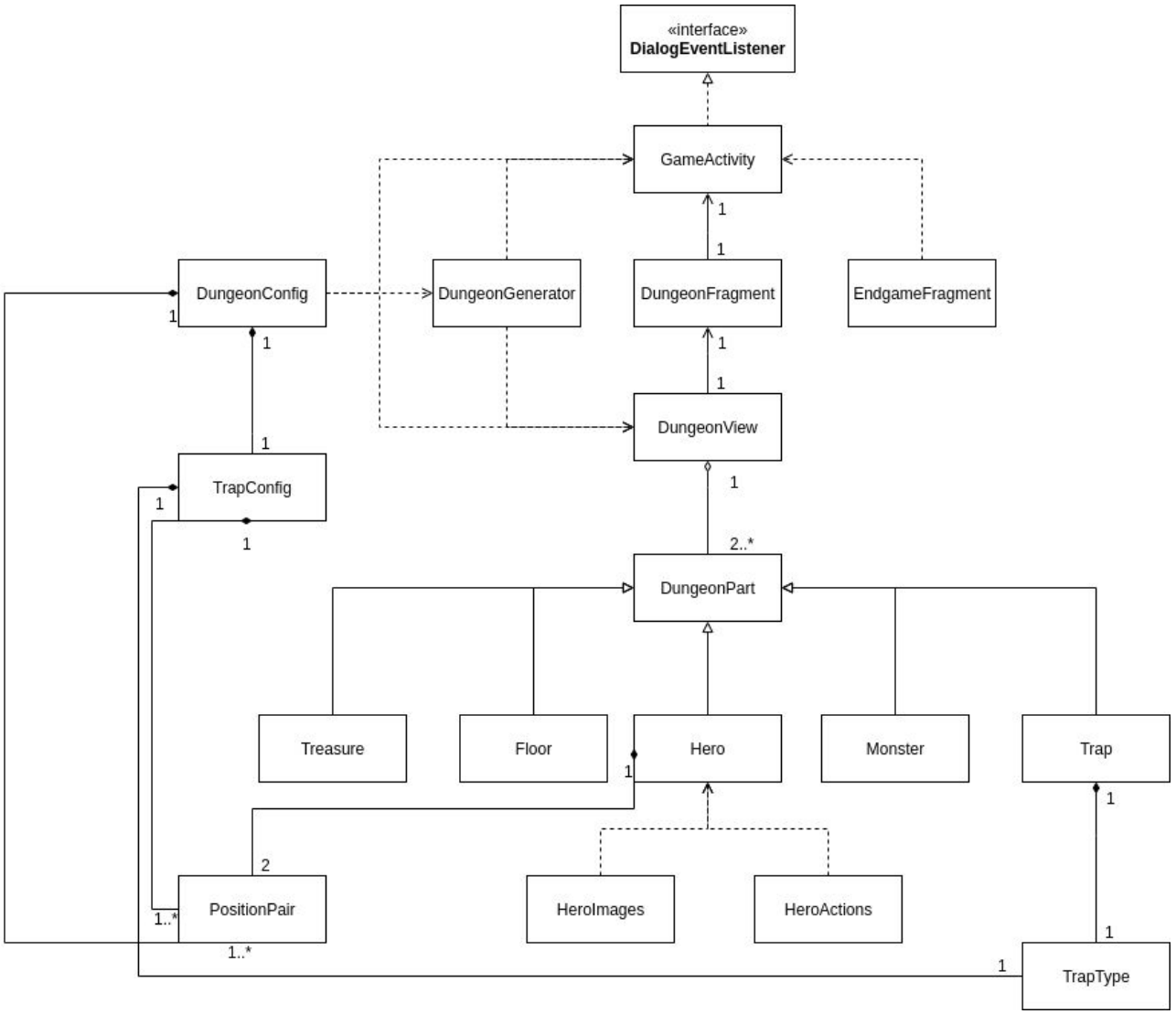

Рисунок 3. UML-диаграмма классов пакета dungeon.

Для полноты картины на UML-диаграмме представлен также класс GameActivity, не входящий в пакет dungeon. Он отвечает за отображение игрового экрана приложения и служит посредником между редактором кода и игровым полем, а также реагирует на важные игровые события (он реализует интерфейс DialogEventListener).

Второй частью игрового режима является редактор кода. Связанные с ним классы расположены в пакете code, их UML-диаграмма показана на Рисунке 4, а список описаний представлен ниже:

1. CodeEditor - интерфейс, определяющий методы для управления редактором кода;
2. CodeFragment - фрагмент, отвечающий за отображение и функционирование редактора кода. Он реализует интерфейс CodeEditor, содержит информацию о типе редактируемой в данный момент программы (для этого определено перечисление ProgramType), а также хранит список команд, доступных для выбора пользователю, и список команд, введенных пользователем и формирующим код программы. Помимо этого в данном классе определяются два важных интерфейса:
    a. HeroMoveListener - данный интерфейс реализует GameActivity, в нем содержатся методы по управлению перемещением игрового персонажа;
    b. InterpreterActionListener - в данном интерфейсе содержатся объявления методов, которые будут вызываться при возникновении важных событий в ходе интерпретации;
3. CommandHolder - подкласс ViewHolder, представляющий переиспользуемый элемент списка команд, доступных для выбора пользователю (списки в приложении реализуются с помощью RecyclerView, соответственно для их отображения необходимо определить подклассы ViewHolder и Adapter). Кроме отображения

элемента списка, он также обрабатывает нажатия на него, инициируя при необходимости отображение вспомогательного диалогового окна, для представления которого могут использоваться классы RepNumPickerFragment или TrapTypePickerFragment;

4. CommandAdapter - подкласс Adapter, являющийся ключевой частью списка команд, доступных для выбора пользователю. Помимо логики работы со списком, он хранит и управляет информацией об уровне вложенности в текущей программе (подробнее о концепциях созданного для игры языка программирования будет рассказано далее);
5. CodeHolder - подкласс ViewHolder, представляющий переиспользуемый элемент списка команд, формирующих код текущей программы. Он управляет отображением строки кода и содержит информацию о тексте команды и номере строки в программе;
6. CodeAdapter - подкласс Adapter, представляющий адаптер для списка команд, формирующих код текущей программы;
7. CommandListItem - класс, содержащий следующую информацию об элементе списка команд, доступных для выбора пользователю: идентификатор изображения команды, хранящегося, как и все изображения в приложении, в res/drawable, и типе команды, представленным значением типа перечисления CommandType;
8. CodeLine - класс, содержащий необходимую для интерпретации программы информацию о строке кода в списке выбранных пользователем команд, а также формирующий текстовое представление команды для ее отображения на экране;
9. Interpreter - класс, представляющий интерпретатор - важнейшую часть приложения, связывающую набранный пользователем код с игровым персонажем. Для корректной обработки циклов был введен вспомогательный класс RepeatConfig, для корректной обработки

подпрограммы - класс Frame, а для корректного позиционирования игрового персонажа - перечисление HeroDirection;

10. Parser - класс, представляющий синтаксический анализатор для проверки корректности набранной пользователем программы перед интерпретацией.

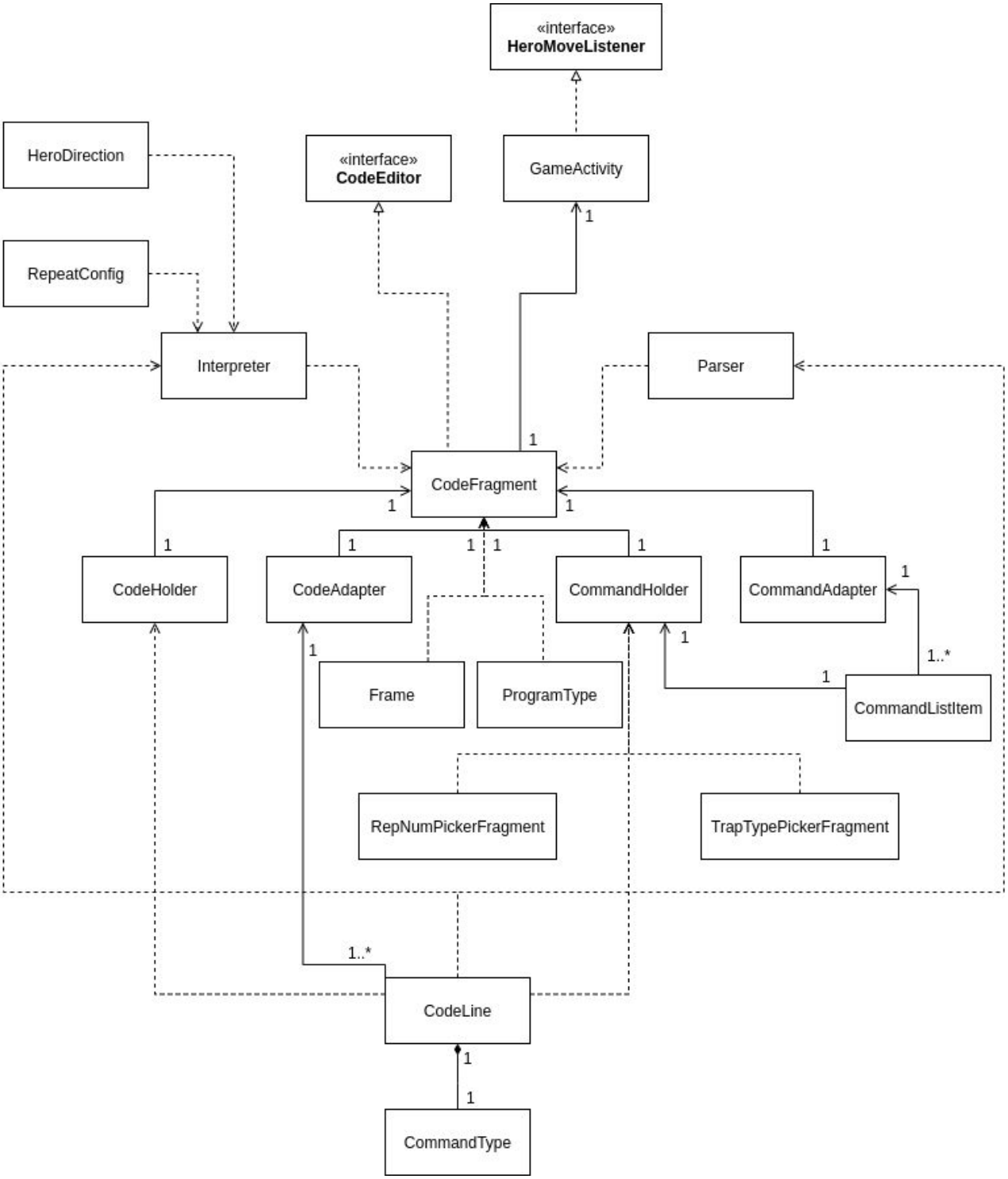

Рисунок 4. UML-диаграмма классов пакета code.

2.2. Проектирование классов режима конструктора

Часть функциональности приложения, соответствующая режиму конструктора уровней, сосредоточена в пакете constructor.

UML-диаграмма данных классов показана на Рисунке 5, а список их описаний представлен ниже:

1. ConstructorFragment - фрагмент, отвечающий за часть экрана, в которой расположено игровое поле, где пользователь способен с помощью нажатия пальцем размещать игровые объекты. Его представление состоит из специально созданного для данного режима подкласса View - ConstructorView. Также фрагмент предоставляет интерфейс для взаимодействия с ConstructorView;
2. ConstructorView - подкласс View, управляющий созданием и отрисовкой игровых объектов, как и DungeonView, но делающий это несколько иным образом. Также он способен удалять последний добавленный объект, полностью очищать игровое поле и формировать объект DungeonConfig в соответствии с текущим состоянием игрового поля;
3. ConstructorPart - наследник абстрактного класса DungeonPart, представляющий игровой объект, добавленный пользователем на поле. Он содержит информацию о типе объекта, представленную значением типа перечисления ConstructorPartType, позиционные данные и картинку с изображением объекта;
4. ConstructorEventListener - интерфейс, определяющий набор методов для управления конструктором уровней. Данный интерфейс реализует класс ConstructorActivity, который не входит в пакет constructor (все Activity приложения расположены в пакете activities). ConstructorActivity служит для отображения двух составляющий конструктора уровней - редактора и игрового поля, а также для организации взаимодействия этих частей и

сохранения полученного от ConstructorView объекта DungeonConfig в виде JSON-файла в Internal Storage;

5. EditorFragment - фрагмент, отвечающий за отображение редактора и за обработку его событий, заключающуюся в делегировании обязанностей классу, реализующему интерфейс ConstructorEventListener (в данном случае это - ConstructorActivity).

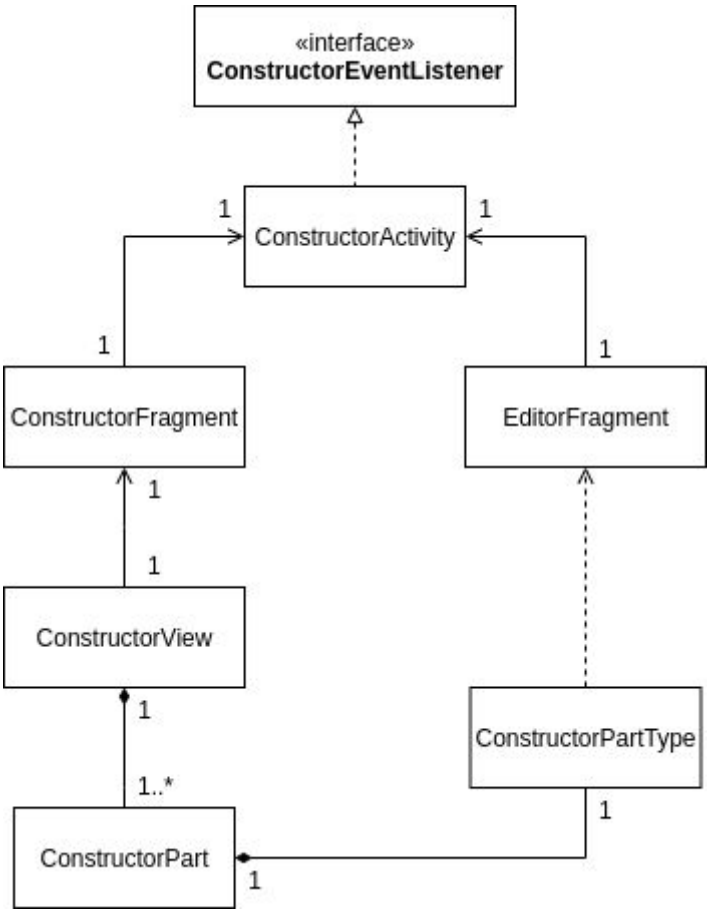

Рисунок 5. UML-диаграмма классов пакета constructor.

2.3. Проектирование оставшейся части приложения

Для того чтобы связать два основных режима приложения - игровой и конструктор уровней, потребовалось создание StartActivity - подкласса Activity, запускающегося первым при старте приложения и содержащего в представлении две кнопки - перехода в режим конструктора уровней и перехода к спискам уровней (предустановленных и созданных самим пользователем).

При переходе к спискам уровней открывается LevelsActivity - подкласс Activity, содержащий два пронумерованных списка уровней. При нажатии на кнопку с названием уровня происходит переход к GameActivity, которой с помощью Intent передается идентификатор уровня, чтобы GameActivity смогла получить по нему нужный конфигурационный JSON-файл.

Логика работы с файловой системой Android, а именно логика получения JSON из Assets и из Internal Storage, и сохранения JSON в Internal Storage, вынесена в специальный класс-репозиторий JsonRepo. Для обеспечения единственности экземпляра данного класса в рамках всего приложения был создан собственный подкласс Application (Application - это класс, являющийся неотъемлемой частью каждого Android-приложения, представляющий собой синглтон, содержащий глобальную информацию о процессе [11]), в котором и была сохранена ссылка на экземпляр JsonRepo.

Также для отладочных целей был создан класс Logger, содержащий набор методов и констант для удобной записи информации в логи [12].

3. Реализация приложения

3.1. Визуализация поля в игровом режиме

Визуализацию игрового поля можно разделить на несколько этапов. На начальном этапе пользователь, находясь на LevelsActivity (см. Рисунок 6), нажимает на кнопку с названием уровня. При этом происходит открытие GameActivity с передачей объекта Intent с информацией о запущенном уровне (идентификатор и булево значение для определения принадлежности к пользовательским уровням).

Рисунок 6. Пользовательский интерфейс LevelsActivity.

На следующем шаге GameActivity с помощью извлеченных из Intent значений обращается к JsonRepo и получает объект String с JSON-строкой, являющейся конфигурацией игрового уровня. Содержание данной конфигурационной строки предельно просто: для каждой игровой сущности в ней хранятся номер строки и столбца на игровом поле, где эта сущность будет расположена (для ловушки еще дополнительно хранится ее тип - левая/верхняя/правая/нижняя). Способы извлечения из файловой системы Android конфигурационных файлов для предустановленных и пользовательских

уровней различны в силу того, что предустановленные уровни хранятся в Assets, а пользовательские - в Internal Storage.

Далее, используя DungeonGenerator и библиотеку GSON, JSON-строка с конфигурацией уровня преобразуется в объект DungeonConfig, содержание которого аналогично JSON-строке (позиционная информация хранится в объектах PositionPair, тип ловушки - в TrapType).

На следующем этапе происходит создание игровых объектов в классе DungeonView. Сначала в методе жизненного цикла onLayout, вызываемого после определения размеров View [13], происходит инициирование создания матрицы стен игрового поля с помощью объекта DungeonConfig, полученного на предыдущем шаге (см. Приложение А).

Каждая клетка игрового поля, инициализированная на предыдущем шаге, в качестве позиционной информации содержит данные о количестве пикселей от левого и верхнего краев мобильного устройства до левого/правого и верхнего/нижнего своих краев соответственно. Это используется при инициализации остальных игровых объектов, например при создании объекта игрового персонажа (см. Приложение А).

Создание Bitmap с изображениями игровых объектов происходит с помощью класса BitmapFactory, который позволяет создать Bitmap на основе drawable-ресурса, и статического метода createScaledBitmap класса Bitmap для придания изображению объекта нужного размера (пример для создания изображения сокровища см. в Приложении А) [14, 15].

Отрисовка игровых объектов происходит следующим образом: переопределенный метод onDraw(Canvas canvas) в DungeonView делегирует обязанность по отрисовке игровых объектов им самим, передавая объект canvas. Те в свою очередь, пользуясь методом drawBitmap объекта canvas отрисовывают соответствующую им картинку, опираясь на свою позиционную информацию. Пример игрового поля показан на Рисунке 7.

Рисунок 7. Пример игрового поля.

3.2. Принцип работы редактора кода

Для осуществления манипуляций игровым персонажем необходимо писать код на придуманном специально для игры языке программирования. Данный язык был создан в обучающих целях для демонстрации некоторых базовых возможностей программирования: последовательного выполнения команд, ветвления, циклов, подпрограмм и рекурсии.

В языке для выделения группы команд, выполняющихся в рамках одной инструкции (например, для выделения тела цикла), используется механизм вложенности, т.е. команды выделяются отступами относительно соответствующих управляющих конструкций (в некотором смысле это напоминает язык Python).

В процессе игры пользователь имеет возможность переключаться между тремя окнами редактора кода:

1. Главное окно, соответствующее телу основной программы, именно эту часть кода начинает выполнять интерпретатор;
2. Окно с кодом подпрограммы (в приложении поддерживается одна подпрограмма), куда пользователь может помещать команды,

выполнение которых будет происходить каждый раз, когда интерпретатор встретит в коде основной программы инструкцию вызова данной подпрограммы. Возможен вызов подпрограммы из самой себя для демонстрации рекурсии (хотя для игры это редко может быть полезно);

3. Окно с кодом защитного скрипта, который будет срабатывать всякий раз, когда игровой персонаж попадет на одну клетку с ловушкой. Данная возможность была добавлена в приложение с целью демонстрации ветвления. Ловушка может быть одного из четырех типов (левая/верхняя/правая/нижняя), поэтому пользователь должен указать, от какого типа ловушки ему нужно уклониться (при указании неверного типа, игровой персонаж погибает от ловушки). Соответственно при наличии на карте нескольких ловушек разных типов необходимо использовать ветвление, чтобы при каждом срабатывании скрипта, была выбрана правильная команда уклонения.

Таким образом, язык поддерживает следующие команды для основной программы и подпрограммы:

1. move() - перемещает персонажа на одну клетку в направлении, зависящим от того, в какую сторону смотрит персонаж в данный момент;
2. turnLeft() - поворачивает персонажа влево;
3. turnRight() - поворачивает персонажа вправо;
4. repeat - управляющая конструкция соответствующая циклу, она повторяет инструкции в своем теле (они имеют на один больший уровень вложенности) заданное количество раз. Также имеется возможность писать вложенные циклы;
5. f() - вызов подпрограммы;

При написании защитного скрипта пользователю доступны следующие команды:

1. if - инструкция, с которой начинается ветвление. Условие, при котором интерпретатор выполнит команды в теле if, может задаваться одним из четырех предикатов: trapLeft (истинно, когда персонаж попал на клетку с левой ловушкой), trapTop, trapRight и trapBottom;
2. elif - комбинация else и if (опять же можно увидеть сходство с языком программирования Python);
3. else - инструкция, тело которой выполняется, если персонаж попал на клетку с ловушкой, тип которой не проверялся в условиях ранее;
4. dodgeLeft() - команда для уклонения от левой ловушки;
5. dodgeTop() - команда для уклонения от верхней ловушки;
6. dodgeRight() - команда для уклонения от правой ловушки;
7. dodgeBottom() - команда для уклонения от нижней ловушки;

На Рисунке 8 представлен пользовательский интерфейс редактора кода, состоящий из следующих элементов, перечисляемых слева направо, начиная с нижней части рисунка:

1. Список команд, доступных для выбора пользователю. При нажатии на команду, ее текст добавляется к коду программы, расположенному выше. При нажатии на команду repeat, перед ее добавлением к коду программы появляется диалоговое окно (используется RepNumPickerFragment) для того, чтобы пользователь указал количество повторений тела цикла (см. Рисунок 9). А при нажатии на команды if или elif в защитном скрипте также появляется диалоговое окно (используется TrapTypePickerFragment) для того, чтобы пользователь выбрал тип ловушки для формирования условия (см. Рисунок 10);
2. Окно с визуальным отображением списка команд, формирующих код программы;
3. Область отображения текущего уровня вложенности с двумя кнопками, позволяющими этот уровень повышать и понижать;

4. Кнопка перезагрузки игры: удаляется весь написанный код, уровень вложенности становится равным нулю, персонаж на игровом поле возвращается в свою начальную точку;
5. Кнопка, открывающая окно с защитным скриптом, меняющая список команд, доступных для выбора пользователю и при повторном нажатии возвращающая к главному окну;
6. Кнопка, открывающая окно для редактирования кода подпрограммы и при повторном нажатии возвращающая к главному окну;
7. Кнопка удаления кода в текущем окне и сбросе его уровня вложенности;
8. Кнопка удаления последней добавленной строки кода в текущем окне;
9. Кнопка, начинающая интерпретацию.

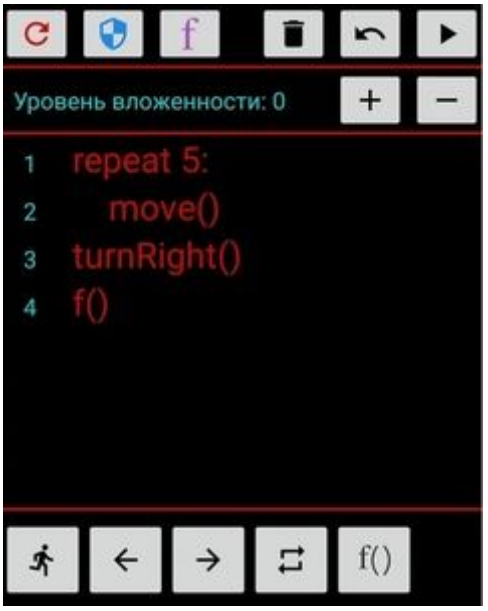

Рисунок 8. Пользовательский интерфейс редактора кода.

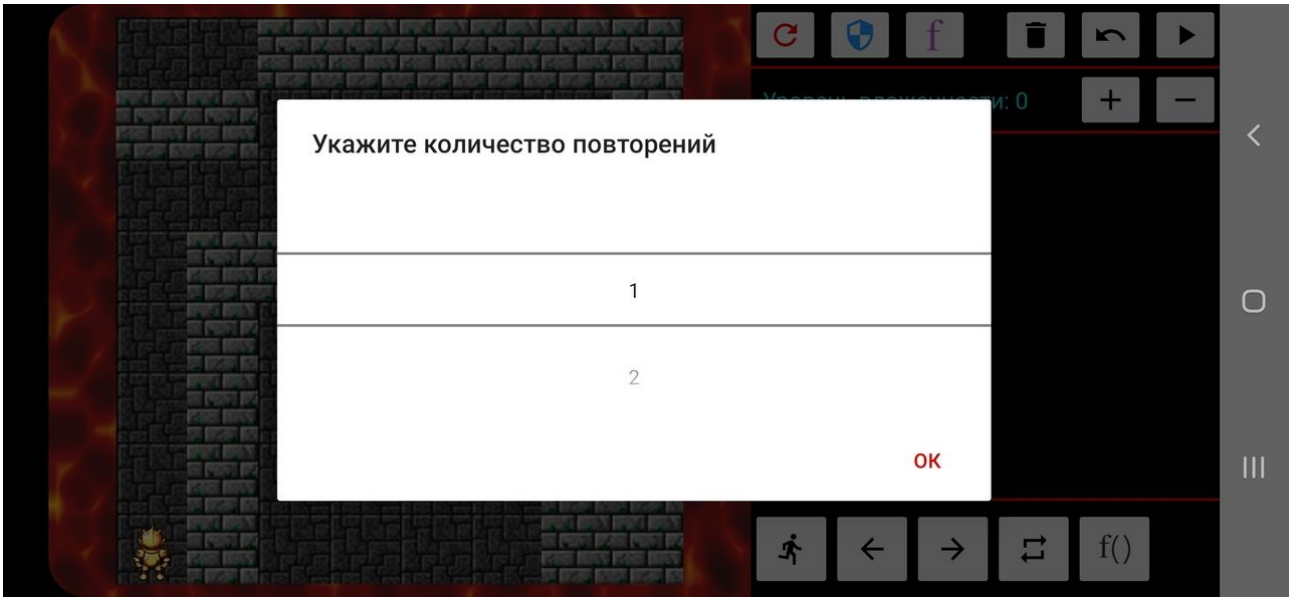

Рисунок 9. Диалоговое окно при нажатии на команду repeat.

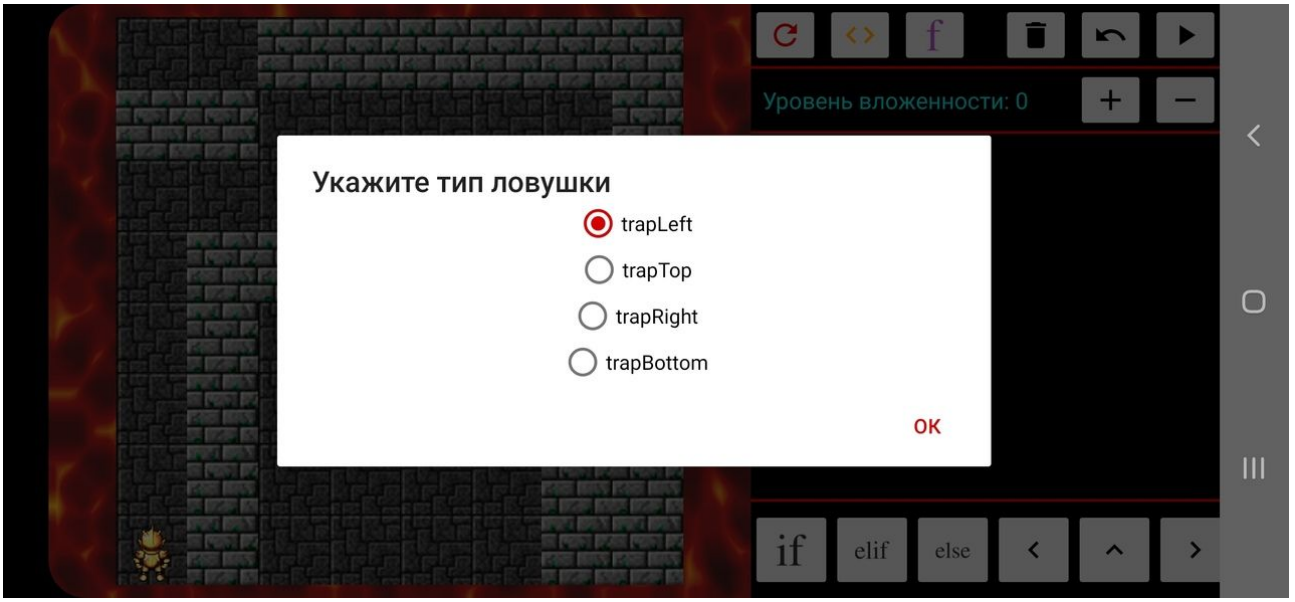

Рисунок 10. Диалоговое окно при нажатии на команды if или elif.

3.3. Связь кода и игрового поля

Для того чтобы персонаж нашел сокровище, пользователь должен написать код, позволяющий герою пройти подземелье, не наткнувшись на скелетов и не попав в ловушки. После написания кода следует запустить интерпретатор.

Однако код начинает интерпретироваться не сразу, сначала происходит этап синтаксического анализа (специальный лексический анализ не нужен в силу того, что код программы представлен в виде списка объектов CodeLine, которые как раз содержат всю необходимую информацию о лексеме языка), на котором программа проверяется на соответствие следующей грамматике, записанной в БНФ, где терминалам соответствуют типы команд, а символ e обозначает пустой символ:

1. Грамматика для основного кода и для кода подпрограммы:

   <Program>::=<PrimCom><Program>|<Repeat><Program>|e

   <PrimCom>::=move()|turnLeft()|turnRight()|f()

   <Repeat>::=repeat <Program>

2. Грамматика для кода защитного скрипта:

   <Program>::=<PrimCom><Program>|<Cond><Program>|e

   <PrimCom>::=dodgeLeft()|dodgeTop()|dodgeRight()|dodgeBottom()

   <Cond>::=if <Program'><Cond'>

   <Program'>::=<PrimCom><Program'>|e

   <Cond'>::=elif <Program'><Cond'>|<Cond''>

   <Cond''>::=else <Program'>|e

Парсеры по этим грамматикам были написаны методом рекурсивного спуска. Стоит отметить, что при анализе они также следили и за уровнем вложенности в программах.

При обнаружении несоответствия написанной программы грамматике, на экран пользователя выводится сообщение об ошибке (см. Рисунок 11) с помощью тоста (класс Toast в Android отвечает за отображение всплывающего уведомления с некоторой информацией [16]).

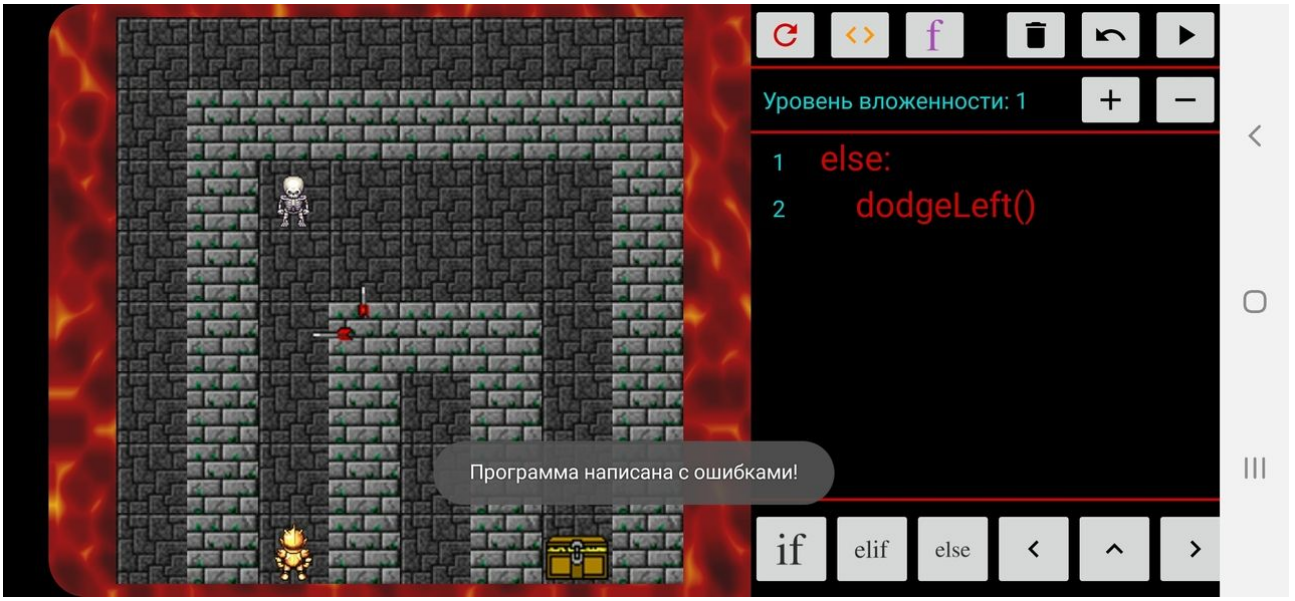

Рисунок 11. Отображение уведомления об ошибке.

Если программа синтаксически корректна, то стартует интерпретация. Общая ее схема такова:

1. Если необработанных инструкций больше нет, то интерпретатор завершает свою работу, иначе считывает очередную необработанную команду;
2. Если данная команда - turnLeft() или turnRight(), то интерпретатор меняет значение переменной, отвечающей за текущее направление игрового персонажа, и продолжает интерпретацию. Он способен влиять на игровое поле через переменную типа HeroMoveListener (данный интерфейс реализует класс GameActivity, и именно он является посредником между редактором кода и игровым полем);
3. Если данная команда - move(), то все становится сложнее, потому что нельзя просто вызвать соответствующий метод у переменной типа HeroMoveListener и продолжить интерпретацию. Дело в том, что персонаж на игровом поле в течение одной секунды совершает анимированное перемещение из одной клетки в другую, поэтому интерпретатор перед выполнением следующей инструкции должен дождаться завершения анимации. Для этого, вызывая метод move() у

переменной типа HeroMoveListener, интерпретатор передает в него колбэк, который объект, представляющий игрового персонажа, должен вызвать по завершении анимации. Вызов этого колбэка приведет к продолжению интерпретации с места остановки.

4. Если данная команда - repeat, то интерпретатор вызывает функцию repeat, передавая ей два параметра - mainConfig типа RepeatConfig (в котором содержится вся необходимая для интерпретации цикла информация - количество повторений, уровень вложенности команд тела цикла, номер строки, начальной для тела цикла, и т.д.) и outerConfigs типа Stack<RepeatConfig> для поддержания вложенных циклов. Далее эта функция выполняет действия, во многом аналогичные действиям основной функции интерпретации, за исключением того, что при наступлении конца тела цикла, функция repeat пытается повторить его еще раз, рекурсивно вызывая себя с количеством повторений тела, уменьшенным на единицу;
5. Если данная команда - f(), то происходит запись в специальный стек вызовов объекта Frame, содержащего информацию о коде исполняемой программы и конфигурациях цикла, если вызов подпрограммы осуществляется из его тела. Также в специальный стек сохраняется текущая позиция интерпретатора. Затем происходит интерпретация кода подпрограммы, после которой из стека вызовов восстанавливается контекст интерпретации.

Как было сказано ранее, при попадании персонажу на клетку с ловушкой запускается защитный скрипт. Анализируя его код, интерпретатор пытается понять, сможет ли персонаж уклониться от ловушки. Общая схема интерпретации защитного скрипта такова:

1. На первом шаге интерпретатор пытается считать команду, если у него это не получается, то персонаж не уклонился. Если это не условная

конструкция, то ее тип сравнивается с типом ловушки, при совпадении результат уклонения - успешный. Если же это инструкция if с условием уклонения от нужного типа ловушки, то результат уклонения определяется телом ветвления, иначе переходим на шаг 2;

2. На шаге 2 пропускается тело неудачного ветвления. Если далее идет elif, то в случае совпадения условия с типом ловушки, результат определяется телом elif, иначе повторяем шаг 2. Если далее идет else, то результат определяется его телом. Иначе переходим к шагу 1.

Как было сказано ранее, интерпретатор управляет объектами на игровом поле через класс GameActivity, который в свою очередь через DungeonFragment дает команды DungeonView.

В случае поступления команды move() внутри DungeonView происходит создание объекта HeroActions, содержащего следующие важные колбэки:

1. onMoveAction - колбэк, срабатывающий на каждом шаге анимации. В нашем случае это - вызов метода invalidate() у DungeonView для перерисовки игрового поля, благодаря чему будет реализовано анимированное перемещение персонажа;
2. onMoveEndAction - колбэк, срабатывающий при завершении анимации перемещения. В нашем случае это - вызов метода, продолжающего интерпретацию программы;
3. onMoveToTrapAction, onMoveToMonsterAction, onMoveToTreasureAction, onMoveToWallAction, onMoveToLavaAction - колбэки, срабатывающие при поражении или победе персонажа, они с помощью класса, реализующего интерфейс DialogEventListener, инициируют показ EndgameFragment;
4. dodgeAction - колбэк, срабатывающий при попадании персонажа на одну клетку с ловушкой, запускает интерпретацию защитного скрипта.

Затем вызывается соответствующий метод у объекта, представляющего игрового персонажа, которому в качестве параметра передается сформированный объект HeroActions. Далее производится попытка определить новую позицию игрового персонажа, если данная позиция находится за пределами поля или на ней расположена стена, то вызываются соответствующие колбэки у объекта HeroActions, и игра заканчивается неудачей. Иначе производится связывание персонажа с игровой клеткой, расположенной на вычисленной позиции (см. Приложение А).

В ходе связывания запускается анимация перемещения персонажа до его новой клетки. Реализуется она с помощью встроенного во фреймворк Android класса ValueAnimator, который позволяет задать начальное и конечное значение, а также время анимации, в ходе которой будет производиться интерполирование значений. Кроме того, для объекта ValueAnimator можно установить обработчик событий обновления значений на каждом шаге анимации [17]. Именно в этом обработчике осуществляется изменения позиционных значений персонажа, изменение картинки с его изображением (чтобы моделировать реалистичную ходьбу главного героя), вызов соответствующих колбэков из HeroActions и старт анимации ловушки, если персонаж перемещается на ее клетку.

В случае, если персонажу не нужно перемещаться, а следует просто повернуться влево или вправо, то меняется лишь картинка с его изображением, после чего выполняется перерисовка игрового поля.

Ловушка, представляющая собой стрелу, анимируется следующим образом (см. Приложение А): как и в случае персонажа, создается объект ValueAnimator, в обработчике обновления значений которого происходит обновления позиционной информации и вызов колбэка, перерисовывающего игровое поле, при завершении анимации стрела возвращается в свое исходное положение.

3.4. Реализация режима конструктора уровней

В режим конструктора уровней можно попасть из стартового меню приложения (см. Рисунок 12).

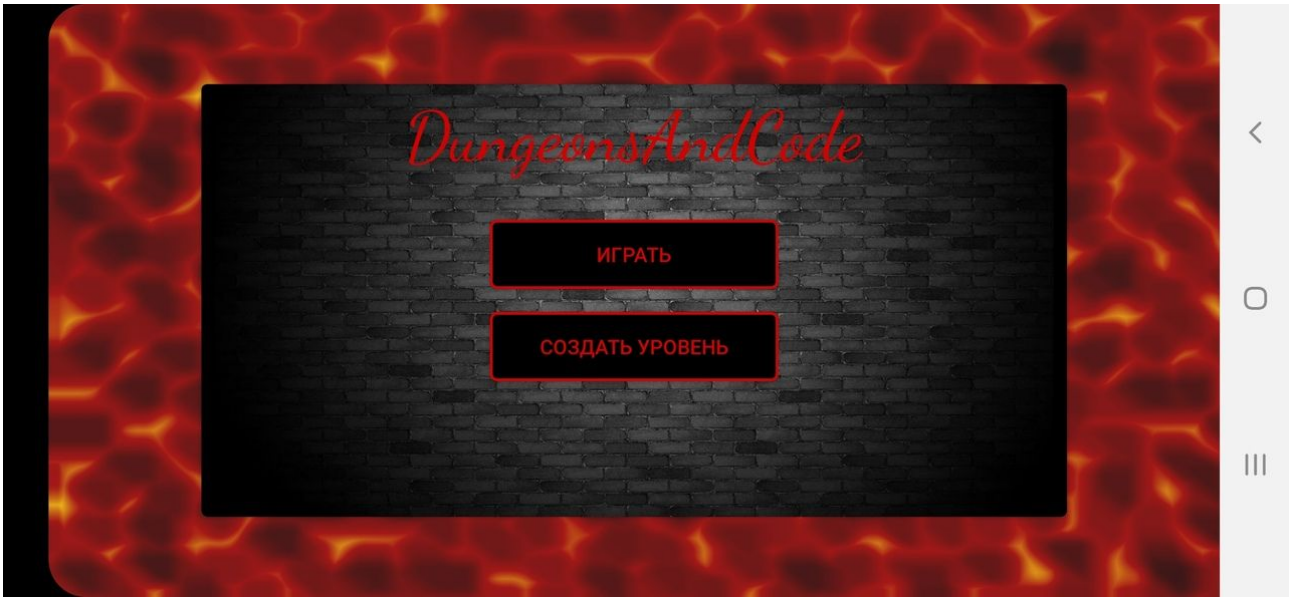

Рисунок 12. Стартовый экран приложения.

Он состоит из игрового поля, где пользователь нажатием пальца может размещать игровые объекты, и редактора, позволяющего управлять созданием уровня (см. Рисунок 13).

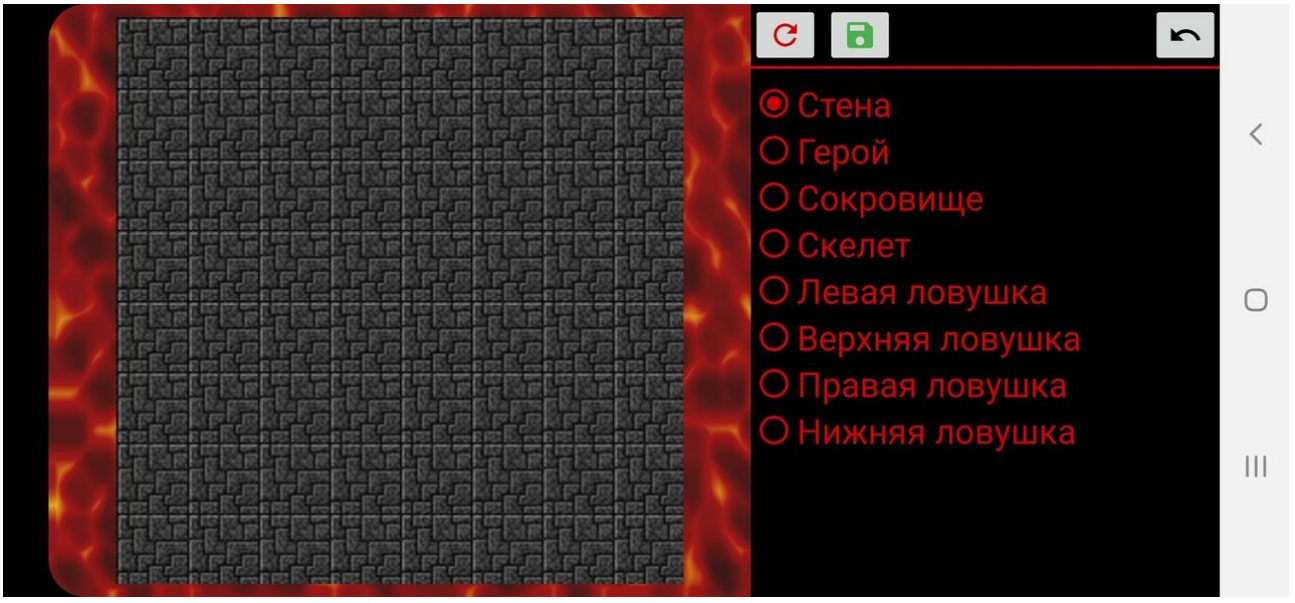

Рисунок 13. Общий вид конструктора уровней.

Представление редактора содержит три кнопки, которые позволяют очистить игровое поле, удалить последний добавленный на поле объект и сохранить конфигурацию уровня в формате JSON, чтобы у пользователя появилась возможность поиграть в собственноручно созданный уровень. Также в представлении редактора имеется виджет RadioGroup, содержащий несколько RadioButton, при нажатии на каждую из которых изменяется тип отрисовываемого объекта, т.е. следующим нажатием на клетку игрового поля пользователь добавит игровой объект уже другого типа.

Класс EditorFragment, отвечающий за представление редактора, обрабатывает события нажатий на кнопки и управляет игровым полем через ConstructorActivity.

Класс ConstructorView, отвечающий за игровое поле, обрабатывает события касаний на клетки поля, в результате которых он создает новые игровые объекты в соответствии с текущим типом отрисовываемой сущности и перерисовывает игровое поле. Кроме того, он поддерживает стек с историей добавления элементов для обеспечения возможности их удаления и метод для получения объекта DungeonConfig, соответствующего текущей матрице игровых объектов.

Непосредственно осуществление сохранения объекта DungeonConfig в Internal Storage происходит в классе ConstructorActivity с использованием метода из JsonRepo.

4. Тестирование приложения

В процессе создания приложения проводились постоянные проверки корректности его работы. Для этого использовались два виртуальных устройства с помощью эмулятора, встроенного в IDE Android Studio, и одно физическое, их характеристики описаны в Таблице 1.

Таблица 1. Характеристики тестовых устройств.

Тип устройства	Название	Размер экрана	Разрешение экрана	Версия Android
Физическое	Samsung Galaxy A50	6,4’’	2340x1080	9.0
Виртуальное	Pixel	5,0’’	1080x1920	7.1.1
Виртуальное	Nexus S	4,0’’	480x800	6.0

Тестирование режима конструктора уровней проводилось следующим образом: проверялась корректность отображения всех игровых сущностей на поле с соблюдением нескольких важных условий:

1. Каждый игровой объект, который можно добавить на поле (т.е. стена, герой, сокровище, скелет или ловушка) должен находиться в отдельной клетке игрового поля;
2. На поле может быть лишь один герой и только одно сокровище;
3. При нажатии пользователем не на клетку игрового поля, никакой объект добавляться не должен.

Далее тестировалось удаление объектов с игрового поля, при этом проверялось, что функция удаления работает корректно даже в случае отсутствия игровых объектов.

Наконец, при попытке пользователем сохранить конфигурацию созданного уровня сначала проверялось обязательное наличие на игровом поле персонажа и сокровища. В случае успешной проверки конфигурация уровня

преобразовывалась в JSON-строку, корректность которой проверялась с помощью специального отладочного класса Logger. Далее проводилось сохранение данной JSON-строки в Internal Storage. Результат успешного сохранения был виден на экране с LevelsActivity, в списке пользовательских уровней которой появлялся новый элемент.

После этого проводился старт созданного уровня, и начиналось тестирование игрового режима. Изначально проверялась правильность отображения на игровом поле сущностей в соответствии с позиционными данными конфигурационного файла.

Далее проводилась проверка работоспособности редактора кода, опираясь на следующие требования:

1. Необходимо, чтобы при нажатии на кнопки с командами, текст данных команд корректно отображался в списке с кодом программы в соответствии с текущим уровнем вложенности. В случае некоторых специальных команд (if, elif, repeat) должно появляться диалоговое окно с возможностью выбора дополнительного параметра инструкции;
2. Должна иметься возможность свободного управления текущим уровнем вложенности в программе;
3. Переключения между основной программой, подпрограммой и защитным скриптом должны быть корректны, т.е. должна происходить смена списка кода, смена списка команд, смена уровня вложенности;
4. Необходимо обеспечение возможностей перезагрузки игры, удаления кода программы в открытом окне с обнулением уровня вложенности, удаления последней добавленной строки кода.

Следующим шагом проверялась работа парсера. Для этого составлялись различные программы, удовлетворяющие и не удовлетворяющие рассмотренным выше грамматикам (например, программы с хаотичными

уровнями вложенности команд, программы с наличием else без if и т.д.), для которых проверялся результат работы синтаксического анализатора.

Затем рассматривалась работа интерпретатора на различный сценариях, демонстрирующих те или иные возможности программирования. При этом обращалось внимание на анимированное перемещение персонажа в соответствии с кодом программы, анимацию ловушки в нужный момент и корректную обработку важных игровых событий, например уклонение от ловушки, нахождение сокровища и т.д. Были также приняты меры по корректной обработке ситуации, когда пользователь решает запустить программу, которая уходит в бесконечную рекурсию.

Тестирование игрового режима при запуске предустановленного уровня тестировалось аналогичным образом. Отличие между запуском предустановленного и пользовательского уровня заключается лишь в том, что конфигурационный файл первого считывается из Assets, а второго - из Internal Storage.

Заключение

Разработана игра под ОС Android, позволяющая пользователям познакомиться с некоторыми базовыми идеями программирования.

В ходе выполнения работы был реализован алгоритм визуализации игрового поля с использованием конфигурационного JSON-файла и алгоритм интерпретации написанного пользователем кода, управляющий анимированным перемещением некоторых игровых объектов.

В игровом режиме был создан пользовательский интерфейс, позволяющий свободно писать и редактировать игровой код, а также запускать интерпретатор.

В режиме конструктора уровней была реализована возможность создания собственного игрового поля с сохранением его конфигурации для дальнейшей игры.

В результате тестирования установлено, что приложение удовлетворяет всем требованиям работы, сформулированным при проектировании.

Список использованных источников

1. Android Developers. Activities.
   URL: https://developer.android.com/guide/components/activities.html?hl=ru
   (дата обращения 17.12.2019)
2. Android Developers. Fragments.
   URL: https://developer.android.com/guide/components/fragments (дата
   обращения 17.12.2019)
3. Codepath. Creating and Using Fragments.
   URL: https://guides.codepath.com/android/creating-and-using-fragments (дата
   обращения 17.12.2019)
4. Филлипс Б., Стюарт К., Марсикано К. Android. Программирование для
   профессионалов. 3-е изд. — СПб.: Питер, 2019. — 687 с.
5. Android Developers. App resources overview. URL:
   https://developer.android.com/guide/topics/resources/providing-resources
   (дата обращения 17.12.2019)
6. Android Developers. Custom View Components.
   URL: https://developer.android.com/guide/topics/ui/custom-components (дата
   обращения 17.12.2019)
7. Android Developers. Create a List with RecyclerView.
   URL: https://developer.android.com/guide/topics/ui/layout/recyclerview (дата
   обращения 17.12.2019)
8. Medium. Android RecyclerView.
   URL: https://medium.com/@droidbyme/android-recyclerview-fca74609725e
   (дата обращения 17.12.2019)
9. Gson. Official GitHub Repository.
   URL: https://github.com/google/gson (дата обращения 17.12.2019)
10. Android Developers. Data and file storage overview.
    URL: https://developer.android.com/training/data-storage (дата обращения
    17.12.2019)
11. Android Developers. Application.
    URL: https://developer.android.com/reference/android/app/Application (дата
    обращения 17.12.2019)
12. Android Developers. Write and View Logs with Logcat.
    URL: https://developer.android.com/studio/debug/am-logcat (дата обращения
    17.12.2019)

13. Code and Trick. Android View Lifecycle.
URL: https://codentrick.com/android-view-lifecycle/ (дата обращения 17.12.2019)
14. StartAndroid. Урок 157. Рисование. Bitmap. BitmapFactory. URL: https://startandroid.ru/ru/uroki/vse-uroki-spiskom/364-urok-157-risovanie-bitmap-bitmapfactory-chtenie-vyvod-na-kanvu-osnovnaja-informatsija.html (дата обращения 17.12.2019)
15. StartAndroid. Урок 158. Рисование. Bitmap. Методы createBitmap. URL: https://startandroid.ru/ru/uroki/vse-uroki-spiskom/365-urok-158-risovanie-bitmap-metody-createbitmap-rabota-s-pikselami-density-mutable.html (дата обращения 17.12.2019)
16. Android Developers. Toast Overview.
URL: https://developer.android.com/guide/topics/ui/notifiers/toasts?hl=ru (дата обращения 17.12.2019)
17. Android Developers. ValueAnimator. URL: https://developer.android.com/reference/android/animation/ValueAnimator (дата обращения 17.12.2019)

Приложение А

Листинг 1. Создание матрицы стен игрового поля.

```
private static final int PADDING_MIN_SIZE = 24;
    static final int FLOORS_ROW_COUNT = 8;

    private DungeonConfig mDungeonConfig;
    private Floor[][] mFloors;
    //прочие поля

    @Override
    protected void onLayout(boolean changed, int left, int top, int right, int bottom) {
        super.onLayout(changed, left, top, right, bottom);
        mFloors = createFloors(Math.min(getHeight(), getWidth()));
        // остальные инициализации
    }

    private Floor[][] createFloors(int globalSize) {
        int floorSize = (globalSize - PADDING_MIN_SIZE * 2) / FLOORS_ROW_COUNT;
        int paddingTop = (getHeight() - floorSize * FLOORS_ROW_COUNT) / 2;
        int paddingLeft = (getWidth() - floorSize * FLOORS_ROW_COUNT) / 2;
        int coordXValue = paddingLeft;
        int coordYValue = paddingTop;

        Floor[][] floors = new Floor[FLOORS_ROW_COUNT][FLOORS_ROW_COUNT];

        boolean[][] isWalls = DungeonGenerator.generateWallsMap(mDungeonConfig);

        for (int i = 0; i < FLOORS_ROW_COUNT; i++) {
            for (int j = 0; j < FLOORS_ROW_COUNT; j++) {
                floors[i][j] = new Floor(
                        coordXValue,
                        coordYValue,
                        coordXValue + floorSize,
                        coordYValue + floorSize,
                        getResources(),
                        isWalls[i][j]
                );

                coordXValue += floorSize;
            }

            coordXValue = paddingLeft;
            coordYValue += floorSize;
        }

        return floors;
    }
```

Листинг 2. Создание объекта игрового персонажа.

```
private Hero createHero() {
        int heroRowPos = mDungeonConfig.getHeroPosition().getRowPosition();
        int heroColPos = mDungeonConfig.getHeroPosition().getColumnPosition();

        Floor startFloor = mFloors[heroRowPos][heroColPos];

        return new Hero(
                startFloor.getLeft(),
                startFloor.getTop(),
                startFloor.getRight(),
                startFloor.getBottom(),
                getResources(),
                mDungeonConfig.getHeroPosition()
        );
}
```

Листинг 3. Создание Bitmap с изображением сокровища.

```
@Override
protected Bitmap createBackgroundImage(Resources resources) {
    return Bitmap.createScaledBitmap(
            BitmapFactory.decodeResource(resources, R.drawable.treasure)
            , mRight - mLeft
            , mBottom - mTop
            , false
    );
}
```

Листинг 4. Реализация анимированного перемещения персонажа.

```
  private void bindWithFloor(Floor[][] floors, final HeroActions heroActions, final HeroImages heroImages) {
    final Floor heroFloor = floors[mCurrentPosition.getRowPosition()][mCurrentPosition.getColumnPosition()];

    PropertyValuesHolder propertyTop = PropertyValuesHolder
        .ofInt(ANIMATION_PROPERTY_TOP, mTop, heroFloor.mTop);
    PropertyValuesHolder propertyRight = PropertyValuesHolder
        .ofInt(ANIMATION_PROPERTY_RIGHT, mRight, heroFloor.mRight);
    PropertyValuesHolder propertyBot = PropertyValuesHolder
        .ofInt(ANIMATION_PROPERTY_BOT, mBottom, heroFloor.mBottom);
    PropertyValuesHolder propertyLeft = PropertyValuesHolder
        .ofInt(ANIMATION_PROPERTY_LEFT, mLeft, heroFloor.mLeft);

    ValueAnimator heroMoveAnimator = new ValueAnimator();
    heroMoveAnimator.setValues(propertyTop, propertyRight, propertyBot, propertyLeft);
    heroMoveAnimator.setDuration(ANIMATION_DURATION);

    heroMoveAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
```

```
    private int animPosition = 0;
    private int frequencyCounter = 0;

    @Override
    public void onAnimationUpdate(ValueAnimator animation) {
      if (
          (int) animation.getAnimatedValue(ANIMATION_PROPERTY_TOP) == heroFloor.mTop &&
              (int) animation.getAnimatedValue(ANIMATION_PROPERTY_RIGHT) == heroFloor.mRight &&
              (int) animation.getAnimatedValue(ANIMATION_PROPERTY_BOT) == heroFloor.mBottom &&
              (int) animation.getAnimatedValue(ANIMATION_PROPERTY_LEFT) == heroFloor.mLeft
      ) {
        mBackgroundImage = heroImages.getStartImage();

        if (heroFloor.getTrap() != null && !heroActions.isDodge(heroFloor.getTrap().getTrapType())) {
          heroActions.moveToTrapAction();
        } else if (heroFloor.hasMonster()) {
          heroActions.moveToMonsterAction();
        } else if (heroFloor.hasTreasure()) {
          heroActions.moveToTreasureAction();
        } else {
          heroActions.moveEndAction();
        }
      } else if (frequencyCounter++ % ANIMATION_FREQUENCY_FACTOR == 0) {
        animPosition = (animPosition + 1) % heroImages.getAnimationImages().size();
        mBackgroundImage = heroImages.getAnimationImages().get(animPosition);
      }

      mTop = (int) animation.getAnimatedValue(ANIMATION_PROPERTY_TOP);
      mRight = (int) animation.getAnimatedValue(ANIMATION_PROPERTY_RIGHT);
      mBottom = (int) animation.getAnimatedValue(ANIMATION_PROPERTY_BOT);
      mLeft = (int) animation.getAnimatedValue(ANIMATION_PROPERTY_LEFT);

      heroActions.moveAction();

      if (heroFloor.getTrap() != null && !heroFloor.getTrap().isAnimated()) {
        heroFloor.getTrap().setAnimated(true);
        heroFloor.getTrap().startAnimation(heroActions.getOnMoveAction());
      }
    }
  });

  heroMoveAnimator.start();
}
```

Листинг 5. Реализация анимации ловушки.

```
private void startAnimation(
    final MoveAction trapMoveAction, final int startValue, final int endValue
) {
    PropertyValuesHolder property = PropertyValuesHolder.ofInt(
            ANIMATION_PROPERTY,
            startValue, endValue
    );

    ValueAnimator trapAnimator = new ValueAnimator();
    trapAnimator.setValues(property);
    trapAnimator.setDuration(ANIMATION_DURATION);

    trapAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {

        @Override
        public void onAnimationUpdate(ValueAnimator animation) {
            int currentValue = (int) animation.getAnimatedValue(ANIMATION_PROPERTY);

            if (currentValue == endValue) {
                setValue(startValue);
                setAnimated(false);
            } else {
                setValue(currentValue);
            }

            trapMoveAction.moveCallback();
        }
    });

    trapAnimator.start();
}

private void setValue(int value) {
    switch (mTrapType) {
        case LEFT:
        case RIGHT:
            mLeft = value;
            break;
        case TOP:
        case BOTTOM:
            mTop = value;
            break;
    }
}
```