СОДЕРЖАНИЕ

[Введение 3](#_Toc113707487)

[1 Общие сведения об ГУО«Гомельский областной центр технического творчества детей и молодёжи» 4](#_Toc113707488)

1.1 История организации...............................................................................4

1.2 Охрана труда и техника безопасности на рабочем месте.....................6

[2 Обзор разработки 2*D*-игр на различных платформах 9](#_Toc113707489)

2.1 Игровой жанр «*Roguelike*»......................................................................9

2.2 Платформа *Godot*...................................................................................10

2.3 Платформа *RPG* *Maker*..........................................................................12

2.4 Платформа *Unity*....................................................................................14

2.5 Разработка на *Unity*2*D*...........................................................................18

3 Разработка игровой механики инвентаря в *Unity*2*D* 21

3.1 Реализация скриптовой базы игровой механики инвентаря...............21

3.2 Реализация отображаемых игровых объектов на сцене......................24

3.3 Верификация игровой механики инвентаря в *Unity*2*D*.......................27

[Заключение 30](#_Toc113707491)

[Список использованных источников 31](#_Toc113707492)

[Приложение А Листинг приложения «*Inventory*» 32](#_Toc113707493)

# ВВЕДЕНИЕ

Преддипломная практика – это обязательная составляющая образовательного процесса, необходимая для подготовки квалифицированных работников, хорошо ориентирующихся не только в профильной теории, но и в реалиях трудовых будней. Этот этап обучения обычно осуществляется вне стен вуза – на базе учреждений, соответствующих будущей специальности студента.

Место практики: ГУО «Гомельский областной центр технического творчества детей и молодёжи».

В ходе обучения важно придерживаться согласованности теории и практики. Как и любой другой этап образовательного процесса, пребывание студента на производстве преследует определенные цели и задачи.

Цели преддипломной практики: закрепление, расширение, углубление и систематизация теоретических знаний, а также приобретение навыков проектирования и конструирования информационных систем.

Задачи преддипломной практики:

– развитие и закрепление практических навыков выполнения анализа предметной области;

– приобретение практического опыта проектирования программных систем;

– развитие и закрепление практических навыков использования языков и инструментальных средств моделирования при проектировании системы;

– развитие и закрепление практических навыков создания программных систем с использованием современных сред разработки, поддерживающих возможность командной работы, контроля проекта и версий системы;

– развитие и закрепление практических навыков разработки документации к системе (технического задания, инструкций пользователя и программиста).

Индивидуальным заданием от предприятия является разработка тестового игрового приложения c реализацией инвентаря на движке *Unity*.

# 1 ОБЩИЕ СВЕДЕНИЯ ОБ ГУО «ГОМЕЛЬСКИЙ ОБЛАСТНОЙ ЦЕНТР ТЕХНИЧЕСКОГО ТВОРЧЕСТВА ДЕТЕЙ И МОЛОДЁЖИ»

1. **История организации**

Государственное учреждение образования «Гомельский областной центр технического творчества детей и молодёжи» создано в 1960 году. Учредитель – главное управление образования гомельского областного исполнительного комитета. Основная деятельность – дополнительное образования детей и молодёжи. В июне 2021 года учреждение сменило юридический адрес и является арендатором помещения технопарка «Коралл». Количество сотрудников – 31 человек, из них 23 ­– педагогических работника, имеющих преимущественно 1 и высшую педагогическую категорию. 3 работника поощрены специальной премией президента по поддержке талантливой молодёжи, один работник является обладателем гранта президента Республики Беларусь.

Количество обучающихся – от 900 до 1000 человек. Ежегодна учащиеся центра являются победителями республиканских и международных конкурсов и соревнований. По итогам 2021 года учреждение занесено на республиканскую доску почёта [1].

На базе учреждения образования реализуется инновационный проект «ИННОПАРК – детский технопарк». Создано многоуровневая непрерывная система образования в области робототехники, электроники, информационных технологий и естественно – математического профиля. Такая система образования представляет собой современный научно-образовательный комплекс, который служит платформой для получения детьми инженерного образования и даёт им мощную базу для поступления в технические ВУЗы, имея при этом определенную специальную подготовку. Логотип учреждения представлен на рисунке 1.1.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1.1 – Логотип учреждения

С 2021 года на базе учреждения реализуется проект онлайн обучения. Открыты 2 областные онлайн школы: *IQUP* и школа точных наук для одарённых учащихся.

Пройти обучение в бесплатной областной дистанционной школе точных наук по математике, физике, информатике, биологии могут ребята 5–8 классов любой школы самого отдаленного уголка Гомельской области. Чтобы быть зачисленными в Школу, необходимо пройти отбор.

В отборе могут участвовать учащиеся – победители, призеры и участники районных, областных, республиканских, международных образовательных мероприятий по естественно-математическому направлению, имеющие отметку за год по выбранному предмету не ниже 7 баллов.

Преимущественным правом на зачисление в Школу пользуются учащиеся из районов Гомельской области (на оставшиеся после отбора места зачисляются учащиеся г. Гомеля). Завершение обучения в Школе признается успешным при условии освоения образовательной программы в полном объеме за период обучения и защиты проекта (прохождения тестирования). По окончании обучения в Школе учащийся получает сертификат. Сертификат окончания обучения представлен на рисунке 1.2.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1.2 – Сертификат об окончании обучения

В областной онлайн школе *IQUP* обучение проходит по инновационным направлениям:

– 3*D* моделирование;

– *Web-Design*;

– *Gamedev* (создание компьютерных игр);

– языки программирования *Phyton, Java, C#;*

*–* английский язык.

В конце учебного года учащиеся онлайн школ проходят офлайн сессии в рамках девятидневных круглосуточных профильных лагерей для одарённых учащихся.

В 2021 году запущен проект онлайн обучения взрослых.

Особое место в учреждении занимает обучение учащихся направлению «Информационные технологии». В 2019 году в детском технопарке открыт отдел «Информационные технологии», в котором представлено 28 направлений для ребят *IT* профиля. Важную составляющую отдела представляют направления движения *Junior Skills Belarus*. По итогам вышеуказанного конкурсного движения, учащиеся центра занимают первое место в республике. Охват учащихся составляет 500 человек в возрасте от 6 лет до 21 года. Заведующий отдела – Жирикова Елена Петровна, которая так же является одним из кураторов практики. Самым востребованным у подростков является создание компьютерных игр.

1. **Охрана труда и техника безопасности на рабочем месте программиста**

Охрана труда – система обеспечения безопасности жизни и здоровья работающих в процессе трудовой деятельности, включающая правовые, социально-экономические, организационные, технические, психофизиологические, санитарно-гигиенические, лечебно-профилактические, реабилитационные и иные мероприятия и средства [2].

Руководством компании уделяется большое внимание улучшению эргономики рабочих мест, обеспечению гигиены и совершенствованию организации труда, регламентации режимов труда и отдыха.

Вся деятельность в области охраны труда на предприятии регламентирована действующим законодательством Республики Беларусь, санитарными нормами и правилами, гигиеническими нормативами, предписаниями надзорных органов.

Важным фактором создания безопасных условий труда является оптимизация организации рабочих мест. Рабочее место, хорошо приспособленное к трудовой деятельности инженера, правильно и целесообразно организованное в отношении пространства, формы, размера, обеспечивает ему удобное положение при работе и высокую производительность труда при наименьшем физическом и психическом напряжении.

При правильной организации рабочего места производительность труда инженера возрастает с 8 до 20 процентов.

Рабочее место для выполнения работ в положении сидя организуется в соответствии с ГОСТ 12.2.032-78 «Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования» [3].

Согласно ГОСТ 12.2.032-78, конструкция рабочего места и взаимное расположение всех его элементов должно соответствовать антропометрическим, физическим и психологическим требованиям. Большое значение имеет также характер работы. В частности, при организации рабочего места программиста должны быть соблюдены следующие основные условия:

* оптимальное размещение оборудования, входящего и состав рабочего места;
* достаточное рабочее пространство, позволяющее осуществлять все необходимые движения и перемещения;
* уровень акустического шума не должен превышать допустимого значения.

Рабочая поза сидя вызывает минимальное утомление программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Помещения для работы программиста должны иметь естественное и искусственное освещение.

Искусственное освещение в помещениях эксплуатации ВДТ и ПЭВМ должно осуществляться системой общего равномерного освещения. В административно-общественных помещениях, в случаях преимущественной работы с документами, допускается применение системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов).

Освещенность на поверхности стола в тоне размещения рабочего документа должна быть 300-500 лк. Местное освещение не должно создавать бликов на поверхности экрана и увеличивать освещенность экрана более 300 лк.

В качестве источников света при искусственном освещении должны применяться преимущественно люминесцентные лампы.

Конструкция рабочего стола должна обеспечивать оптимальное размещение на рабочей поверхности используемого оборудования с учетом его количества и конструктивных особенностей (размер ВДТ и ПЭВМ, клавиатуры и др.), характера выполняемой работы. При этом допускается использование рабочих столов различных конструкций, отвечающих современным требованиям эргономики.

Конструкция рабочего стула (кресла) должна обеспечивать поддержание рациональной рабочей позы при работе, позволять изменять позу с целью снижения статического напряжения мышц шейно-плечевой области и спины для предупреждения развития утомления.

Тип рабочего стула (кресла) должен выбираться в зависимости от характера и продолжительности работы.

Рабочий стул (кресло) должен быть подъемно-поворотным и регулируемым по высоте и углам наклона сиденья и спинки, а также расстоянию спинки от переднего края сиденья, при этом регулировка каждого параметра должна быть независимой, легко осуществляемой и иметь надежную фиксацию.

Экран видеомонитора должен находиться от глаз пользователя на оптимальном расстоянии 600-700 мм, но не ближе 500 мм с учетом размеров алфавитно-цифровых знаков и символов.

Высота рабочей поверхности стола должна регулироваться в пределах 680-800 мм, при отсутствии такой возможности высота рабочей поверхности стола должна составлять 725 мм.

Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной – не менее 500 мм, глубиной на уровне колен – не менее 450 мм и на уровне вытянутых ног – не менее 650 мм.

Клавиатуру следует располагать на поверхности стола на расстоянии не менее чем 300 мм от края, обращенного к пользователю или на специальной, регулируемой по высоте рабочей поверхности, отделенной от основной столешницы.

Помимо требований к организации рабочего места СанПиН 9-131 РБ 2000 устанавливает требования к микроклимату рабочей зоны: влажности, температуре, скорости потока воздуха и пр.

Техника безопасности – это система организационных мероприятий и технических средств, предотвращающих воздействие на работающих опасных производственных факторов.

# ОБЗОР РАЗРАБОТКИ 2*D*-ИГР НА РАЗЛИЧНЫХ ПЛАТФОРМАХ

**2.1 Игровой жанр «*Roguelike*»**

Разработка игры в жанре «*Roguelike*» требует многих навыков и инструментов. Некоторые из основных этапов разработки включают в себя идею, проектирование, разработку, тестирование и выпуск игры. Одним из ключевых инструментов для разработки 2*D*-игр в жанре «*Roguelike*» являются игровые движки, такие как *Unity*, *Godot* и другие. Эти инструменты позволяют разработчикам создавать игровые объекты, уровни, механику игры и многое другое.

Чтобы создать эффективную систему, разработчики должны учитывать множество факторов, таких как размер уровня, количество врагов, размещение предметов и многое другое. Также важно убедиться, что уровни создают интересный и разнообразный игровой процесс.

2*D*-игры в жанре «*Roguelike*» обычно имеют вид сверху вниз и основаны на традиционных играх-лабиринтах. Игроки должны исследовать уровни, бороться с врагами, собирать предметы и выживать, чтобы дойти до конца игры. Ключевой особенностью игр в жанре «*Roguelike*» является то, что каждая игра уникальна, стиль, избранный разработчиком, может разительно отличаться от других подобных игр, ввиду обширности и разнообразности подходящих для жанра «*Roguelike*» тем. Пример игры такого жанра приведён на рисунке 2.1.



Рисунок 2.1 – 2*D*-игра в жанре «*Roguelike*»

Хотя 2*D* игры в жанре «*Roguelike*» могут быть довольно сложными и требовательными, их глубокий игровой процесс, частая реиграбельность и приятный глазу визуальный стиль делают их одними из самых увлекательных игр на рынке. Разработчики продолжают создавать новые и захватывающие игры в этом жанре, которые продолжают привлекать игроков со всего мира и удивлять их своей глубиной и сложностью.

2.2 Платформа *Godot*

*Godot* – это бесплатный игровой движок с открытым кодом, который подходит для создания 2*D* и 3*D* игр различных жанров и масштабов. Он имеет свой язык программирования - *GDScript*, который схож с *Python*, но с меньшим числом функций. В *Godot* также можно использовать *C*++, *C*# и другие языки программирования.

Одной из основных преимуществ *Godot* является его простота и удобство использования. Он имеет интуитивно понятный интерфейс, а также инструменты для быстрого создания игровых объектов и сцен. Godot также имеет множество инструментов и функций для упрощения разработки игр, таких как интегрированный редактор сцен, механизм анимации, физический движок, система частиц, искусственный интеллект и т.д.

*Godot* поддерживает различные платформы, включая *Windows*, *macOS*, *Linux*, *Android*, *iOS*, *Web* и другие. Это позволяет разработчикам создавать игры для различных устройств и платформ, что увеличивает их охват и аудиторию. Пример рабочего интерфейса *Godot* представлен на рисунке 2.2.

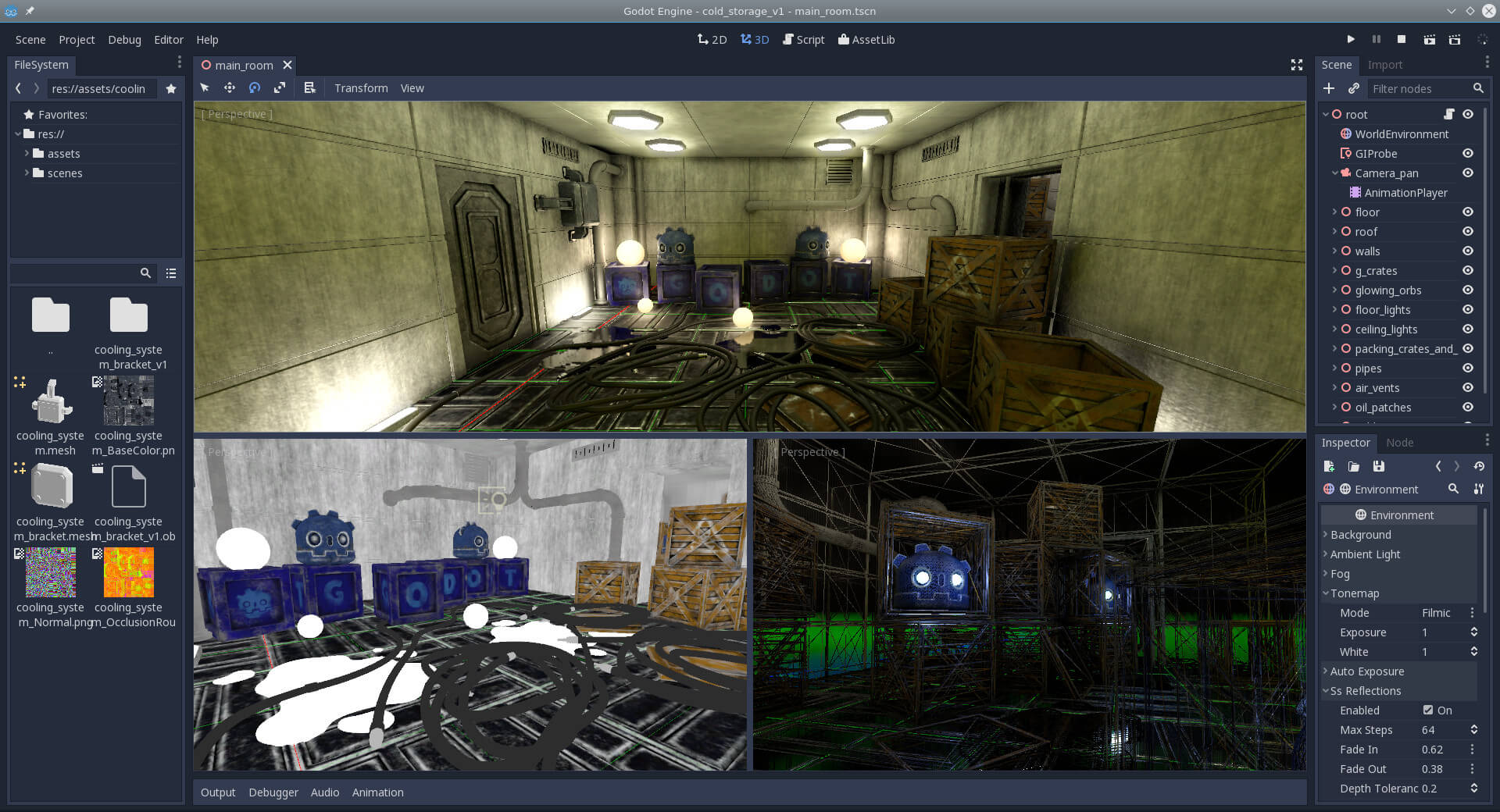


Рисунок 2.2 – Интерфейс игрового движка *Godot*

*Godot* также имеет активное сообщество, где можно получить поддержку и решить возникшие проблемы. Сообщество разработчиков *Godot* регулярно обновляет движок, добавляя новые функции и улучшая производительность.

Однако, на данный момент, *Godot* не имеет такой большой популярности как *Unity* или *Unreal* *Engine*, что может означать, что в некоторых случаях, может быть сложнее найти подрядчиков или решить проблемы во время разработки.

Хотя *Godot* не так популярен, как *Unity* или *Unreal* *Engine*, некоторые известные игры были разработаны на этом движке. Вот несколько из них:

1. «*Baba Is You*» – это паззл-игра, разработанная финским разработчиком Арво Партаненом на *Godot*. Игрок должен изменять правила игры, чтобы достичь цели.
2. «*Vigil: The Longest Night*» – это мрачная 2*D*-игра жанра «*Metroidvania*» с элементами хоррора. Она была разработана студией *Glass* *Heart* *Games* на *Godot*.
3. «*MewnBase*» – это симулятор выживания на Луне, где игрок должен создавать и управлять своей базой, добывать ресурсы и выживать. Она была разработана Крисом Килианом на *Godot*.
4. «*FAR: Lone Sails*» – это постапокалиптическая игра-головоломка, где игрок управляет транспортным средством, преодолевает препятствия и исследует мир. Она была разработана студией *Okomotive* на *Godot*.

Среди всех игр, созданных на Godot, особое внимание заслужила *Deponia*, а точнее её портированые на *PlayStation 4*/*iOS* версии. *Deponia* – это серия приключенческих игр, разработанных и выпущенных немецкой студией *Daedalic* *Entertainment*. Пример версии для *iOS* представлен на рисунке 2.3.



Рисунок 2.3 – Версия игры *Deponia* написанная на *Godot*

Графика в игре выполнена вручную, в качестве основного инструмента разработки был использован движок *Visionaire Studio*, позволяющий создавать классические приключенческие игры с точки зрения разработки. В *Deponia* сочетается юмор и драма, игра получила положительные отзывы за свой геймплей, визуальный стиль и замысловатый сюжет.

Хотя этот список не исчерпывает все игры, сделанные на *Godot*, эти игры показывают, что *Godot* может быть использован для создания игр различных жанров и масштабов.

* 1. Платформа *RPG* *Maker*

*RPG Maker* – это игровой движок, предназначенный для создания классических ролевых игр. Он был создан компанией *Enterbrain* в Японии и был выпущен в 1988 году. С тех пор множество версий были выпущены, каждая с улучшенными функциями и возможностями. Движок имеет простой и понятный пользовательский интерфейс, который позволяет даже новичкам создавать игры без необходимости в знании программирования. *RPG* *Maker* включает в себя редакторы миров, персонажей, монстров, событий, диалогов и битв, что позволяет создавать различные сценарии и настройки для игры. Пример интерфейса *RPG Maker* предоставлен на рисунке 2.4.



Рисунок 2.4 – Интерфейс игрового движка *RPG Maker*

*RPG* *Maker* имеет большое сообщество пользователей, которые создают и делятся своими играми и ресурсами. Есть множество бесплатных и платных DLC, которые добавляют новые функции и ресурсы, такие как текстуры, музыка и звуки. Также существует огромное количество бесплатных скриптов и плагинов, которые расширяют возможности движка.

Самыми распространёнными и популярными играми на игровом движке *RPG Maker* являются:

1. «*Corpse* *Party*» – это хоррор-игра, созданная с помощью *RPG* *Maker* и разработанная *Team* *GrisGris*. Игроки управляют группой школьников, застрявших в потустороннем мире, где они должны бороться за свою жизнь, искать способы выбраться из этого места и раскрыть тайну, которая привела их сюда. Игра использует мрачную и атмосферную графику и эффекты, чтобы создать ужасающую атмосферу.
2. «*Yume* *Nikki*» – это экспериментальная игра-исследование, созданная и изданная *Kikiyama*. Игрок управляет девочкой по имени *Madotsuki*, которая проводит большую часть своего времени внутри своих собственных снов. Игроки исследуют различные лабиринты, миры и фантазии *Madotsuki*, сталкиваясь с различными существами и узнавая о ее личности и истории. Игра получила высокую оценку за свою уникальную и атмосферную визуальную стилистику.
3. «*To the Moon*» – это приключенческая игра, разработанная и изданная фирмой *Freebird* *Games*. Игроки следуют за Джонни и Ривер, двумя докторами, которые специализируются на изменении воспоминаний людей перед их смертью, чтобы они могли исполнить свои заветные мечты. Игроки путешествуют через различные моменты жизни Джонни и Ривер, решая головоломки и исследуя их взаимоотношения. Игра была высоко оценена за свою уникальную историю и красивую графику. Пример данной игры представлен на рисунке 2.5.



Рисунок 2.5 – Игра «*To the Moon*», созданная на движке *RPG Maker*

Одним из основных преимуществ *RPG* *Maker* является его доступность. Даже если у вас нет опыта в программировании или дизайне игр, вы можете легко создать свою игру с помощью этого инструмента. Кроме того, благодаря огромному сообществу пользователей и разработчиков, вы всегда можете получить помощь и поддержку в создании своей игры.

Это лишь несколько примеров игр, созданных с помощью *RPG* *Maker*, которые показывают широкий диапазон жанров и стилей игр, которые можно создавать с помощью этого инструмента.

Хотя *RPG* *Maker* может быть ограничен в сравнении с другими игровыми движками в терминах графики и функциональности, он все еще является одним из самых популярных и доступных инструментов для создания классических ролевых игр.

* 1. Платформа *Unity*

*Unity* – это мощный инструмент для разработки игр, который объединяет в себе разнообразные инструменты, необходимые для создания игрового контента. Этот движок позволяет разработчикам быстро и легко создавать игры, при этом обладая полным контролем над процессом разработки.

Благодаря обширной документации и сообществу разработчиков, *Unity* является одним из самых популярных инструментов для создания игр на сегодняшний день. Его графический интерфейс интуитивно понятен и удобен в использовании, что позволяет создавать игры различных жанров и сложности. *Unity* также обеспечивает высокую производительность, что делает его идеальным выбором для создания игр для мобильных устройств, консолей и ПК.

В первую очередь, движок *Unity* дает возможность разрабатывать игры, не требуя для этого каких-то особых знаний. Здесь используется компонентно-ориентированный подход, в рамках которого разработчик создает объекты (например, главного героя) и к ним добавляет различные компоненты (например, визуальное отображение персонажа и способы управления им).

Благодаря удобному *Drag* *&* *Drop* интерфейсу и функциональному графическому редактору движок позволяет рисовать карты и расставлять объекты в реальном времени и сразу же тестировать получившийся результат.

Такой подход дает возможность быстро и эффективно прототипировать игры, а также ускоряет процесс создания уровней и сцен в проекте. Все это делает *Unity* одним из самых удобных и популярных игровых движков на сегодняшний день.

Основными преимуществами *Unity* являются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов. К недостаткам относят появление сложностей при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек.

Движок поддерживает два скриптовых языка: *C*# и *JavaScript*. Расчёты физики производит физический движок *PhysX* от *NVIDIA*. Среду разработки *Unity*, можно увидеть на рисунке 2.6.



Рисунок 2.6 – Среда разработки *Unity*

На *Unity* написаны тысячи игр, приложений и симуляций, которые охватывают множество платформ и жанров. При этом *Unity* используется как крупными разработчиками, так и независимыми студиями. Также преимуществом движка является наличие огромной библиотеки ассетов и плагинов, с помощью которых можно значительно ускорить процесс разработки игры. Их можно импортировать и экспортировать, добавлять в игру целые заготовки – уровни, врагов, паттерны поведения искусственного интеллекта и др. Многие ассеты доступны бесплатно, другие предлагаются за небольшую сумму, и при желании можно создавать собственный контент, публиковать его в *Unity Asset Store* и получать от этого прибыль.

Проект в *Unity* делится на сцены (уровни) – отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе как, собственно, объекты (модели), так и пустые игровые объекты – объекты, которые не имеют модели («пустышки»). Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название, может быть тег (метка) и слой, на котором он должен отображаться. Так, у любого объекта на сцене обязательно присутствует компонент *Transform* – он хранит в себе координаты местоположения, поворота и размеров объекта по всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент *Mesh* *Renderer*, делающий модель объекта видимой. К объектам можно применять коллизии, которых существует несколько типов.

Также *Unity* поддерживает физику твёрдых тел и ткани, а также физику типа *Ragdoll* (тряпичная кукла). В редакторе имеется система наследования объектов; дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов.

При импорте текстуры в *Unity* можно сгенерировать *alpha*-канал, *mip*-уровни, *normal*-*map*, *light*-*map*, карту отражений, однако непосредственно на модель текстуру прикрепить нельзя – будет создан материал, которому будет назначен шейдер, и затем материал прикрепится к модели. Редактор *Unity* поддерживает написание и редактирование шейдеров. Редактор *Unity* имеет компонент для создания анимации, но также анимацию можно создать предварительно в 3*D*-редакторе и импортировать вместе с моделью, а затем разбить на файлы.

*Unity* 3*D* поддерживает систему *Level* *Of* *Detail* (сокр. *LOD*), суть которой заключается в том, что на дальнем расстоянии от игрока высокодетализированные модели заменяются на менее детализированные, и наоборот, а также систему *Occlusion* *culling*, суть которой в том, что у объектов, не попадающих в поле зрения камеры, не визуализируется геометрия и коллизия, что снижает нагрузку на центральный процессор и позволяет оптимизировать проект. При компиляции проекта создается исполняемый (.*exe*) файл игры (для *Windows*), а в отдельной папке – данные игры (включая все игровые уровни и динамически подключаемые библиотеки)

Движок поддерживает множество популярных форматов. Модели, звуки, текстуры, материалы, скрипты можно запаковывать в формат unityassets и передавать другим разработчикам, или выкладывать в свободный доступ. Этот же формат используется во внутреннем магазине *Unity* *Asset* *Store*, в котором разработчики могут бесплатно и за деньги выкладывать в общий доступ различные элементы, нужные при создании игр. Чтобы использовать *Unity* *Asset* *Store*, необходимо иметь аккаунт разработчика *Unity*. *Unity* имеет все нужные компоненты для создания мультиплеера. Также можно использовать подходящий пользователю способ контроля версий. К примеру, *Tortoise* *SVN* или *Source* *Gear*.

В *Unity* входит *Unity* *Asset* *Server* – инструментарий для совместной разработки на базе *Unity*, являющийся дополнением, добавляющим контроль версий и ряд других серверных решений.

Rак правило, игровой движок предоставляет множество функциональных возможностей, позволяющих их задействовать в различных играх, в которые входят моделирование физических сред, карты нормалей, динамические тени и многое другое. В отличие от многих игровых движков, у *Unity* имеется два основных преимущества: наличие визуальной среды разработки и межплатформенная поддержка. Первый фактор включает не только инструментарий визуального моделирования, но и интегрированную среду, цепочку сборки, что направлено на повышение производительности разработчиков, в частности, этапов создания прототипов и тестирования. Под межплатформенной поддержкой предоставляется не только места развертывания (установка на персональном компьютере, на мобильном устройстве, консоли и т. д.), но и наличие инструментария разработки (интегрированная среда может использоваться под *Windows* и *Mac* *OS*);

Другим преимуществом называется модульная система компонентов *Unity*, с помощью которой происходит конструирование игровых объектов, когда последние представляют собой комбинируемые пакеты функциональных элементов. В отличие от механизмов наследования, объекты в *Unity* создаются посредством объединения функциональных блоков, а не помещения в узлы дерева наследования. Такой подход облегчает создание прототипов, что актуально при разработке игры;

В качестве недостатков приводятся ограничение визуального редактора при работе с многокомпонентными схемами, когда в сложных сценах визуальная работа затрудняется. Вторым недостатком называется отсутствие поддержки *Unity* ссылок на внешние библиотеки, работу с которыми программистам приходится настраивать самостоятельно, и это также затрудняет командную работу. Ещё один недостаток связан с использованием шаблонов экземпляров (англ. *prefabs*). С одной стороны, эта концепция *Unity* предлагает гибкий подход визуального редактирования объектов, но с другой стороны, редактирование таких шаблонов является сложным. Также, *WebGL*-версия движка, в силу специфики своей архитектуры (трансляция кода из *C*# в *С*++ и далее в *JavaScript*), имеет ряд нерешённых проблем с производительностью, потреблением памяти и работоспособностью на мобильных устройствах.

На *Unity* написаны сотни игр, приложений и симуляций, *Unity* используется как крупными разработчиками (например, *Blizzard*), так и в создании инди-игр.

Одними из самых успешных и узнаваемых игр, созданных на платформе *Unity*, являются:

– *Cuphead –* игра в жанре «*run and gun*» и платформера, разработанная и изданная канадской командой разработчиков *StudioMDHR Entertainment*;

– *Firewatch* – приключенческая игра, разработанная компанией *Campo* *Santo* и изданная компанией *Panic*;

– *Pillars of Eternity* – ролевая игра, разработанная *Obsidian Entertainment* и выпущенная издательством *Paradox Interactive*.

Пример игры на *Unity* можно увидеть на рисунке 2.7.



Рисунок 2.7 – Игра *Firewatch* на *Unity*

* 1. **Разработка на Unity2D**

Двумерные игры сравнительно просты: для них не требуется сложных 3*D*-моделей, программный код по сравнению с 3*D*-проектами выглядит понятнее. Такие игры популярны как на компьютерах, так и на мобильных устройствах. *Unity* также позволяет разрабатывать игры и для браузеров.

Создание 2*D*-игры с нуля многим легче нежели разработка полноценного 3D-проекта не только из-за отсутствия третьего измерения, но и из-за различия в процессах анимации и обработки некоторых игровых объектов. К тому же пиксельная стилистика, которую чаще всего используют именно в 2*D*-играх, является оптимальным и несложным вариантом создания собственного авторского контента, будь то рисунки заднего фона, персонажей или же предметов и врагов, с которыми игрок взаимодействует.

Предварительно рассмотрим основные понятия *Unity*, без понимания которых будет проблематично создать игру:

– ресурс или *Asset* – основная программная единица для любого проекта *Unity*. Это может быть изображение, трёхмерная или двумерная модель, звук;

– игровой объект или *GameObject*. Если ресурс используется в сцене, то он становится игровым объектом. Например, у нас есть изображение противника – это ресурс;

– компоненты влияют на поведение и отображение игровых объектов;

– префаб (англ. *prefabricated object*) – способ хранения игровых объектов, оптимизированный для многократного использования и клонирования с разными настройками. При изменении префаба изменяются все его копии;

– cкрипт – исходный текст программы на языке *C*#. Могут прикрепляться к игровым объектам или префабам.

Выбор режима 2*D* или 3*D* перед началом определяет некоторые настройки редактора *Unity* – например, будут ли изображения импортированы как текстуры или же как спрайты. В процессе разработки игры существует возможность переключаться между этими режимами без каких-либо последствий. Благодаря такому подходу достигается разнообразие перспектив, а также возможность создавать игры со смешанной графикой, например, 2.5*D*-игры. Пример такой игры приведён на рисунке 2.8.

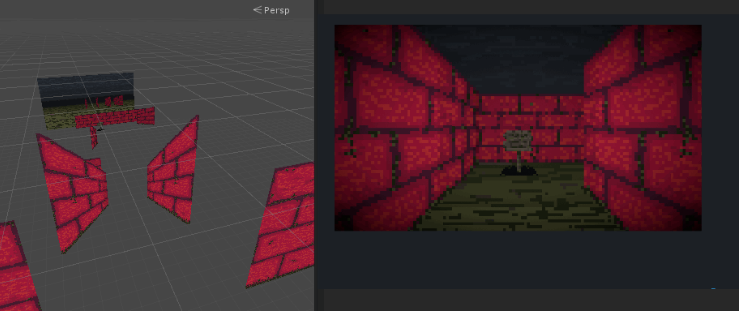


Рисунок 2.8 – 2.5*D*-игра на платформе *Unity*

Игры, использующие 3*D* графику, но при этом применяющие ортографическую камеру вместо перспективы – эта технология используется в играх, где отображение происходит с высоты птичьего полёта, обычно такие игры называются «2.5*D*». Если вы создаёте игру подобную этой, вы должны так же использовать редактор в 3*D* режиме, так как несмотря на отсутствие перспективы, вы будете по-прежнему работать с 3*D* моделями и ассетами. Вам нужно будет лишь переключить ваши камеру и вид сцены в режим *Orthographic*.

В большинстве 2*D* игр персонажи, визуальные эффекты и прочие игровые объекты представляют собой спрайты – двумерные картинки, использующие лишь два измерения: *X* и *Y*. Не имеющие третьего измерения, они всё также являются прекрасными инструментами для реализаций идей разработчиков видеоигр, так как в целом являют собой довольно удобный и быстрый способ добавления в игру определённого рода стиля.

Некоторые игры используют трёхмерные модели для персонажей и окружения, но при этом ограничивают геймплей двумя измерениями. Пример: камера может использовать «боковое скольжение», и игрок может двигаться только в двух измерениях, но игра по-прежнему использует трёхмерные модели в качестве препятствий, а камера имеет перспективу. Для таких игр 3*D* выполняет скорее стилистическую роль, нежели функциональную. Хотя геймплей и двухмерный, вы всё равно будете управлять трёхмерными моделями, поэтому для создания такой игры следует выбирать режим 3*D*, пример такой игры представлен на рисунке 2.9.



Рисунок 2.9 – 2*D* геймплей с трёхмерной графикой

Встроенный в *Unity* физические движки обеспечивают разработчика компонентами для обработки симуляции физики. Благодаря огромной вариативности для настройки свойств и поведения объектов, абсолютно любое желание разнообразить игровой процесс может быть исполнена буквально за пару минут экспериментов с настройками в соответствующей вкладке. Благодаря использованию скриптов, объекту можно придать динамику, например, автомобиля, самолёта, птицы или же подводной лодки, движок способен обеспечивать поддержку большинства игровых решений разработчика.

Игровой жанр игры, для которой будет реализована механика инвентаря является крайне популярным в настоящее время. Сочетание *Roguelike* элементов вместе с элементами *RPG* являются самым оптимальным сочетанием, которое гарантирует большое пространство для реализации различных механик, при этом сохраняя единую идею, которой следует игровой процесс. Геймплей игры, использующих данное сочетание, чаще всего является репетативным или же продолжительным по времени, что позволяет раскрыть потенциал каждого аспекта сочетания.

Для реализации тестового игрового приложения был выбрал игровой движок *Unity*, т.к. он обладает наибольшим инструментарием и информационной базой, которые в один момент являются удобными и простыми.

# РЕАЛИЗАЦИЯ ИГРОВОЙ МЕХАНИКИ ИНВЕНТАРЯ В *UNITY*2*D*

3.1 Реализация скриптовой базы игровой механики инвентаря

В данной главе описана скриптовая реализация игровой механики инвентаря с использованием инструментария *Unity* на языке *Unity* *Sharp*. В результате разработки игровой механики, позволяющей игроку иметь инвентарь, были реализованы скрипты, отвечающие за взаимодействие игрока с предметами, а также взаимодействие предметов с игровым интерфейсом, являющимся основой механики.

В скрипте *Database*.*cs* описана сущность игрового инвентаря, как списка предметов, имеющих собственные параметры. Данная логика описана следующим образом:

– класс *Database* содержит в себе объявление списка переменных типа *Item* под названием *items*. Данная переменная используется для доступа к базе данных игровых предметов, доступных для игрока. Все предметы описаны в игровой базе данных и таким образом являются допустимыми к взаимодействию с логикой инвентаря;

– класс *Item* является описанием игрового предмета. Каждый игровой предмет имеет следующие поля: *int* *id* – номер игрового предмета, *string name* – название игрового предмета, а также *Sprite img* – ссылка на изображение, которое используется в качестве отображаемого спрайта в инвентаре.

В скрипте *Inventory*.*cs* описана основная логика механики инвентаря, связанная с визуальным отображением предметов в инвентаре, появлением предметов в инвентаре при поднятии их игроком, обновлением информации о предметах инвентаря, а также использованием уже находящихся внутри инвентаря предметов игроком. Данная логика описана в следующих методах:

– *public void Start()* – метод, который вызывается при запуске объекта. Он используется для инициализации переменных, установки начальных значений, таких как показатель жизни тестового персонажа и запуска метода *AddGraphics()*;

– *public void Update()* – метод, который вызывается каждый кадр. Он используется для обновления состояния объекта, проверки ввода пользователя и выполнения других действий. В нём описано открытие и закрытие окна инвентаря нажатием клавиши «*I»*, обновление списка предметов в инвентаре на тот случай, если предмет был поднят, пока инвентарь был закрыт, а также случайная генерация предметов в инвентаре, как тест, путём нажатия клавиши «A» на клавиатуре;

– *public void SearchForSameItem(Item item, int count)* – метод, предусмотренный для механики перемещения игровых предметов по инвентарю, таким образом, что при наложении предмета на предмет, они могли сложиться в одну кучу. Данный метод необходим для реализации различных малых предметов, способных к сложению в кучи внутри инвентаря. Переменными, передающимися в данный метод, являются: *item* – тип игрового предмета и *count* – количество игрового предмета;

– *public void* *AddItem(int id*, *Item item*, *int count)* – метод, отвечающий за добавление игрового предмета в инвентарь при его взаимодействии с игроком на уровне. Переменными, передающимися в данный метод, являются *id* – номер предмета в игровой базе данных, *item* – тип игрового предмета и *count* – количество игрового предмета. Тип добавляемого игрового предмета зависит от его названия. После взаимодействия с игроком, игровой предмет добавляется в первую пустую ячейку инвентаря;

– *public void AddGraphics()* – метод, реализующий инициализацию игрового инвентаря в начале игровой сессии. В нём описано заполнение поля инвентаря пустыми ячейками, а также обработка события – нажатия на игровой предмет в инвентаре;

– public void *UpdateInventory()* – данный метод производит обновление информации обо всех предметах инвентаря и используется в составе метода *public void Update()*. Таким образом обновление игрового инвентаря происходит в момент открытия и закрытия инвентаря, а также тестового его заполнения. Благодаря этому методу исключается постоянное обновление инвентаря, что благоприятно влияет на производительность;

– *public void SelectObject()* – метод, вызываемый при нажатии на игровой объект в инвентаре и определяющий событие, происходящее после этого. Таким образом при нажатии на объект типа «зелье» происходит пополнение здоровья игрока и уничтожение предмета «зелье» в данной ячейке инвентаря, тогда как при нажатии на пустую ячейку ничего не происходит;

– *public void UseItem(int id, Item item, int count)* – метод, вызываемый при использовании предмета игроком. После вызова данного метода предмет исчезает из игрового инвентаря, заменяясь на пустую ячейку. Таким образом происходит использование предмета;

– *public void MoveObject()* – метод, привязывающий игровой предмет к мыши игрока. Данный метод используется при перемещении предмета по инвентарю;

– *public ItemInventory CopyInventoryItem(ItemInventory old)* – метод, который заменяет предметы местами при условии, что предмет, который игрок переносит не того же типа, что и предмет, находящийся в ячейке, в которую игрок переносит первый предмет.

В скрипте *Inventory*.*cs* реализован класс *ItemInventory*, содержащий поля *int* *id*, *GameObject* *itemGameObj*, *int count*. Данный класс используется при работе с предметами инвентаря.

Также в скрипте *Inventory*.*cs* присутствуют публичные поля для игровых объектов и переменных следующего типа:

– *Database data* – игровая база данных с описанными в ней игровыми объектами, которые можно добавить в инвентарь игрока;

– *GameObject gameObjShow* – префаб игрового предмета, находящегося в инвентаре;

– *GameObject InventoryMainObject* – область игрового инвентаря, в которой отображаются находящиеся в нём предметы и пустые ячейки;

– *int maxCount –* переменная, определяющая максимальное количество ячеек игрового инвентаря;

– *Camera cam* – игровая камера, к которой привязан интерфейс инвентаря;

– *EventSystem es* – система обработки событий, необходимая для обработки события нажатия на предметы в инвентаре;

– *Text hp* – тестовый текст на экране, имитирующий здоровье игрового персонажа;

– *int currentID* – номер в игровой базе данных предмета, выбранного игроком в инвентаре;

– *ItemInventory currentItem* – предмет, в данный момент выбранный игроком в инвентаре;

– *int currentHP* – переменная показателя здоровья персонажа на данный момент времени;

– *RectTransform movingObject* – переменная координат на сцене для отображения переносимых игроком предметов;

– *Vector3 offset* – переменная сдвига изображения для отображения переносимых игроком предметов;

– *GameObject background* – задний фон игрового инвентаря

В скрипте *PlayerController*.*cs* описана логика взаимодействия тестового игрового персонажа с игровыми предметами на сцене (например зельем). В данном скрипте реализован следующий метод:

– *public void OnTriggerEnter2D(Collider2D collision)* – это метод, который вызывается, когда другой объект с компонентом *Collider*2*D*(зелье) входит в триггер-зону объекта, на котором этот метод находится (тестовый игровой персонаж).

В зависимости от типа игрового объекта, с которым столкнулся игровой персонаж, в игровой инвентарь добавляется тот или иной игровой предмет. Игровой объект, находящийся на сцене, при этом исчезает. Таким образом игрок «поднимает» предмет и помещает его в инвентарь.

Также в скрипте *PlayerController*.*cs* присутствуют поля для игровых объектов типа *Inventory* и *Database*, таким образом игрок через взаимодействие инициализирует создание предметов из заполненной игровой базы данных, при этом обращаясь к интерфейсу инвентаря.

Данная скриптовая база является достаточной и оптимальной для тестирования механики инвентаря.

* 1. Реализация отображаемых игровых объектов на сцене

В результате разработки игровой механики, позволяющей игроку иметь инвентарь, были реализованы различные игровые объекты, наполняющие игровую сцену, на которые затем были наложены скрипты и произведены настройки, связанные с взаимодействием игровых предметов друг с другом.

Игровой объект «*Main Camera*» является игровой камерой, которая отрисовывает тестовую сцену и основной интерфейс игрового инвентаря, таким образом к нему применены скрипты *Database.cs* и *Inventory.cs*. На рисунке 3.1 изображены скрипты, применённые к данному игровому объекту и настроенные для корректной работы.

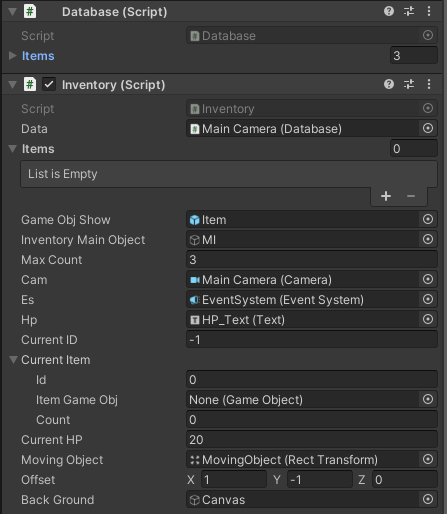


Рисунок 3.1 – Скрипты игрового объекта «*Main Camera*»

Игровой объект «*Canvas*» является основой для интерфейса инвентаря. Благодаря данному игровому объекту у игрока имеется возможность включать и выключать отображение игрового инвентаря.

Игровой объект «*BG*» (*background*) является задним фоном и контейнером для игрового объекта «*MI*» (*main inventory*), позволяющим настраивать размер и форму игрового инвентаря на усмотрение разработчика. Компонент «*Grid Layout Group*» позволяет разделить игровой объект на равные сектора, в которые затем будут помещены префабы игровых ячеек. На рисунке 3.2 изображены связанные игровые объекты «*BG*» – серый цвет – и «*MI*» – зелёный цвет.

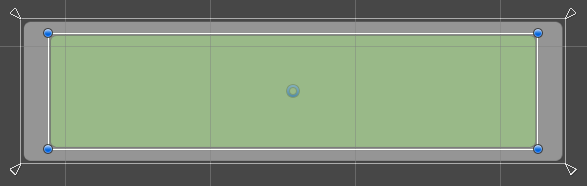


Рисунок 3.2 – Игровые объекты «*BG*» и «*MI*»

Игровой объект «*MI*» является контейнером для списка объектов и префабов типа «*Item*». В пределах области отображения «*MI*» и находятся ячейки инвентаря во время работы тестового приложения.

Игровой объект «*HP\_Text*» является текстовой строкой, отображающей количество очков жизней тестового персонажа и предназначен для теста использования предмета типа «зелье» из игрового инвентаря. На рисунке 3.3 изображён игровой объект «*HP\_Text*», находящийся на игровой сцене.



Рисунок 3.3 – Игровой объект «*HP\_Text*»

Игровой объект «*EventSystem*» добавлен средой разработки *Unity* для обработки во время работы приложения различного рода событий. Примером такого события можно назвать нажатие левой кнопкой мыши по предмету в инвентаре.

Игровой объект «*Test\_Flask*» является тестовым игровым предметом типа «зелье». При появлении игрового предмета типа «зелье» на сцене в консоль пишется сообщение «Зелье на сцене.». Во время взаимодействия с тестовым игровым персонажем также пишется сообщение – «Произошла коллизия.»

Для данного игрового предмета были произведены настройки размеров и формы коллайдера, что позволит ему взаимодействовать с другими игровыми предметами, одним из которых является тестовый игровой персонаж и отображаемого изображения, которое было заменено на изображение тестового зелья. Данные изменения в настройке игрового предмета можно пронаблюдать на рисунке 3.4.

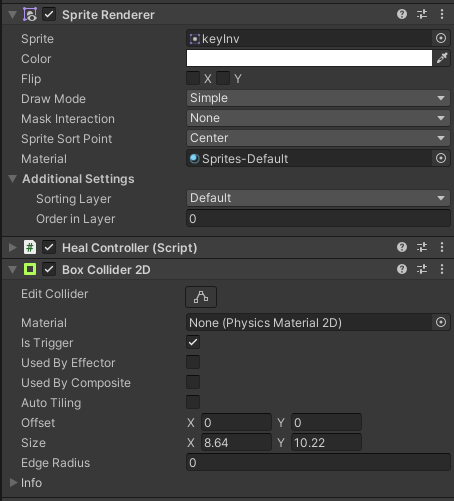


Рисунок 3.4 – Настройки игрового предмета «*Test\_Flask*»

Игровой объект «*Test\_Personage*» является тестовым объектом типа «игрок». Данный игровой объект является отображением игрока, взаимодействующего с игровыми предметами на сцене. Для данного игрового объекта были произведены настройки размеров и формы коллайдера, а также отображения спрайта тестового игрового персонажа на сцене. К игровому объекту «*Test\_Personage*» применён скрипт «*PlayerController*.*cs*», а также привязаны игровая база данных и инвентарь, в который персонаж будет передавать предметы, поднятые на игровой сцене.

Так как здоровье персонажа в первую очередь отображается на игровом объекте «*Canvas*», здоровье тестового игрового персонажа привязано именно к камере, а не к игровому персонажу. Это не влияет на работоспособность данной механики. Основную работу тестовый персонаж производит именно с игровым инвентарём, добавляя в него подобранные при коллизии предметы. Настройки и применённые к игровому объекту «*Test\_Personage*» скрипты изображены на рисунке 3.5.

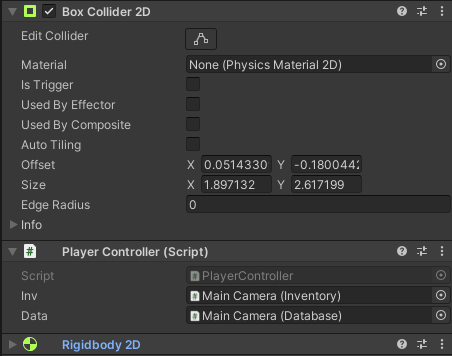


Рисунок 3.5 – Настройка игрового объекта «*Test\_Personage*»

Данные игровые объекты являются достаточными и оптимальными для демонстрации игровой механики инвентаря.

* 1. Верификация игровой механики инвентаря в *Unity*2*D*

Верификация игрового приложения является важным этапом в процессе разработки игр, так как она позволяет обнаружить и исправить ошибки и проблемы, которые могут привести к сбоям, ошибкам и другим нежелательным результатам в процессе эксплуатации приложения.

Таким образом, верификация игрового приложения является важным этапом в процессе разработки игр, который помогает обеспечить качество, удобство использования, безопасность и соответствие требованиям.

При запуске тестового приложения происходит отрисовка игровых объектов, таких как игровой инвентарь, тестовый персонаж, тестовые предметы и т.д. Также инициализируются все игровые скрипты, а также вызываются методы «*Start*» и «*Update*».

Игровой инвентарь сразу же отображает пустые ячейки, а для его заполнения случайными игровыми объектами нужно нажать клавишу «*A*» на клавиатуре.

По итогу начальное окно игрового приложения отображает счётчик жизней персонажа, самого тестового персонажа, несколько игровых предметов (два тестовых зелья и тестовый ключ), а также окно инвентаря, состоящее из нескольких игровых объектов/элементов.

Пример отображения начального окна игрового приложения изображён на рисунке 3.6.



Рисунок 3.6 – Начальное состояние тестовой сцены «*Inventory*»

При перемещении спрайта игрока к предмету и вызове коллизии предмет исчезает с игровой сцены, добавляясь в инвентарь, а в консоль выводятся соответствующие сообщения. Пример такого взаимодействия изображён на рисунке 3.7.

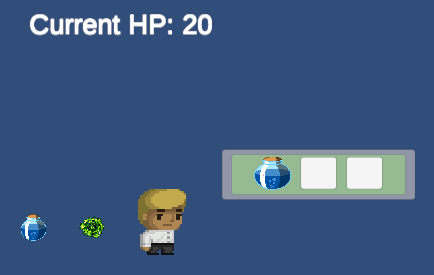


Рисунок 3.7 – Результат коллизии игровых объектов «*Test\_Personage*» и «*Test\_Flask*»

После взаимодействия с предметом, затем перемещённым в игровой инвентарь, проверяется вызов события при нажатии на него в ячейке инвентаря.

После работы скрипта можно увидеть, что предмет исчез из инвентаря, здоровье персонажа изменилось, а в консоль было выведено соответствующее сообщение, рисунок 3.8.

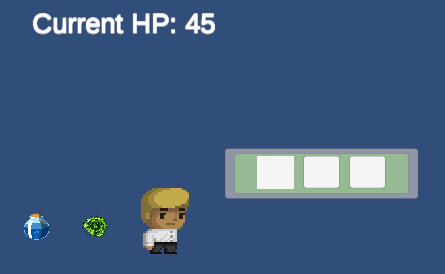


Рисунок 3.8 – Результат нажатия на игровой объект в инвентаре

При взаимодействии персонажа с предметами и их добавлении в инвентарь исключены ошибки, при которых предметы заменяют друг друга. Если в инвентаре есть свободная ячейка, её занимает предмет, который игровой персонаж поднимает в данный момент.

Также при нажатии клавиши «*I*» на клавиатуре игровой инвентарь закрывается, не теряя данных о находящихся в нём предметах.

# ЗАКЛЮЧЕНИЕ

Результатом дипломной работы является компьютерное игровое приложение «*Gone Summer: Overgrowth*» в жанре «*Roguelike*» с элементами *RPG*.

В ходе выполнения работы был проведён анализ уже существующих продуктов, выявлены их достоинства и недостатки, на основе которых и разрабатывалась логика приложения.

В ходе разработки решены следующие задачи:

– произведен аналитический обзор доступных программных средств разработки, позволяющих разработать игровое приложение;

– разработан пользовательский графический интерфейс, реализующий взаимодействие пользователя с инвентарем и инвентарем быстрого доступа;

– разработана общая игровая структура и механики, позволяющие осуществлять игровой процесс;

– применены подходы проектирования на основе *Unity*2*D*, позволяющие разработать масштабируемое приложение;

– произведена верификация и тестирование конечного программного продукта.

Игровое приложение реализовано при помощи использования среды разработки *Unity2D*.

Приложение представляет из себя несколько наполненных интерактивными предметами и врагами игровых локаций-уровней, связанных друг с другом переходами. Пользователь управляет игровым персонажем, который свободно передвигается по игровому миру. Задача и цель игры – пройти все уровни.

Отличительной особенностью игрового приложения от аналогов является наличие разнообразных игровых механиках и уровней.

Тестирование разработанного приложения показало, что игровое приложение выполняет свои функции, игровые механики функционируют должным образом.

Кроме того, игровое приложение является масштабируемым, что позволяет впоследствии внедрять новую функциональность.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [Электронный ресурс] Официальный сайт ГУО «Гомельский Областной Центр Технического Творчества Детей И Молодёжи». Режим доступа: [*http://octt.gomel.by*](http://octt.gomel.by) – Дата доступа: 03.04.2023.
2. [Электронный ресурс]: Юридический словарь. – Режим доступа: *http://multilang.pravo.by/ru/Term/Index/525?langName=ru&ch=Все&size=25&page=4&type=3* – Дата доступа: 03.04.2023.
3. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования: ГОСТ 12.2.032-78. – Введ. 01.01.1979 – Москва: ИПК Издательство стандартов, 2001. – 27с.

4. Хокинг, Дж. *Unity* в действии. Мультиплатформенная разработка на *С*# / Дж. Хокинг ­– Спб: Питер, 2019 – 352с.

5. Ларкович, С. Справочник *UNITY*. Кратко, быстро, под рукой / *С*. Ларгович – Москва: Наука и техника, 2020 – 288с.

6. Роджерс, С. *Level Up! The Guide to Great Video Game Design* – *USA*: *Wiley* / С. Роджерс. – Пер. с англ. – СПб.: Символ-Плюс, 2010 – 125с.

7. Маннинг, Дж. *Unity Game Development Cookbook* / Дж. Маннинг, T.Нуггент – Пер. с англ. – СПб.: Символ-Плюс, 2019 – 400с.

Кожевников, Е.А. Расчёт экономической эффективности разработки программных продуктов: метод. указания по подготовке организационно-эконо-мического раздела дипломных работ для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / Е.А. Кожевников, Н.В. Ермалинская. – Гомель: ГГТУ им. П.О. Сухого, 2012. – 68 с.

9. Кухаренко, С.Н. Электронный учебно-методический комплекс дисциплины «охрана труда» для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / С.Н. Кухаренко, Д.В. Соболев. – Гомель: ГГТУ им. П.О. Сухого, 2013. – 93 с.

# ПРИЛОЖЕНИЕ А

(обязательное)

# Листинг программы «*Inventory*»

**Листинг *Database*.*cs*:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

// Создание списка игровых предметов

public class Database : MonoBehaviour

{

public List<Item> items = new List<Item>();

}

[System.Serializable]

// Класс реализующий игровой предмет

public class Item

{

public int id;

public string name;

public Sprite img;

}

**Листинг *Inventory*.*cs*:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.EventSystems;

using Unity.VisualScripting;

// Основной класс реализующий инветарь

public class Inventory : MonoBehaviour

{

public Database data;

public List<ItemInventory> items = new List<ItemInventory>();

public GameObject gameObjShow;

public GameObject InventoryMainObject;

public int maxCount;

public Camera cam;

public EventSystem es;

public Text hp;

public int currentID;

public ItemInventory currentItem;

public int currentHP;

public RectTransform movingObject;

public Vector3 offset;

public GameObject backGround;

public void Start()

{

if(items.Count == 0)

{

AddGraphics();

}

hp.text = "Current HP: " + currentHP;

}

// Стандартный метод обновления сцены в Unity покадрово

public void Update()

{

if(currentID != -1)

{

MoveObject();

}

if(Input.GetKeyDown(KeyCode.I))

{

backGround.SetActive(!backGround.activeSelf);

if(backGround.activeSelf)

{

UpdateInventory();

}

}

if(Input.GetKeyDown(KeyCode.A))

{

for(int i = 0; i < maxCount; i++ ) //Testing (rand element adding)

{

AddItem(i, data.items[Random.Range(0, data.items.Count)], 1);

}

UpdateInventory();

}

}

// Метод поиска одинаковых предметов в инвентаре

public void SearchForSameItem(Item item, int count)

{

for(int i = 0; i < maxCount; i++)

{

if (items[i].id == item.id)

{

if (items[0].count < 1)

{

items[i].count += count;

if (items[i].count > 1)

{

count = items[i].count - 1;

items[i].count = 1;

}

else

{

count = 0;

i = maxCount;

}

}

}

}

if(count > 0)

{

for(int i = 0; i < maxCount; i++)

{

if (items[i].id == 0)

{

AddItem(i, item, count);

i = maxCount;

}

}

}

}

// Метод добавления игрового предмета в инвентарь

public void AddItem(int id, Item item, int count)

{

if (items[id].itemGameObj.GetComponent<Image>().sprite.name == "UISprite" || items[id].itemGameObj.GetComponent<Image>().sprite.name == "emptyInv")

{

items[id].id = item.id;

items[id].count = count;

items[id].itemGameObj.GetComponent<Image>().sprite = item.img;

if (count > 1 && item.id != 0)

{

items[id].itemGameObj.GetComponentInChildren<Text>().text = count.ToString();

}

else

{

items[id].itemGameObj.GetComponentInChildren<Text>().text = "";

}

}

else

{

AddItem(id + 1, item, count);

}

}

/\*

public void AddInventoryItem(int id, ItemInventory invItem)

{

items[id].id = invItem.id;

items[id].count = invItem.count;

items[id].itemGameObj.GetComponent<Image>().sprite = data.items[invItem.id].img;

if (invItem.count > 1 && invItem.id != 0)

{

items[id].itemGameObj.GetComponentInChildren<Text>().text = invItem.count.ToString();

}

else

{

items[id].itemGameObj.GetComponentInChildren<Text>().text = "";

}

}

\*/

// Метод добавление пользовательского интерфейса

public void AddGraphics()

{

for(int i = 0; i < maxCount; i++)

{

GameObject newItem = Instantiate(gameObjShow, InventoryMainObject.transform) as GameObject;

newItem.name = i.ToString();

ItemInventory ii = new ItemInventory();

ii.itemGameObj = newItem;

RectTransform rt = newItem.GetComponent<RectTransform>();

rt.localPosition = Vector3.zero;

rt.localScale = new Vector3(1, 1, 1);

newItem.GetComponentInChildren<RectTransform>().localScale = new Vector3(1, 1, 1);

Button tempButton = newItem.GetComponent<Button>();

tempButton.onClick.AddListener(delegate { SelectObject(); });

items.Add(ii);

}

}

// Метод обновления данных о предметах, находящихся в инвентаре

public void UpdateInventory()

{

for(int i = 0; i < maxCount; i++)

{

if (items[i].id != 0 && items[i].count > 1)

{

items[i].itemGameObj.GetComponentInChildren<Text>().text = items[i].count.ToString();

}

else

{

items[i].itemGameObj.GetComponentInChildren<Text>().text = "";

}

items[i].itemGameObj.GetComponent<Image>().sprite = data.items[items[i].id].img;

}

}

// Метод, который обрабатывает нажатие кнопки ячейки инвентаря

public void SelectObject()

{

currentID = int.Parse(es.currentSelectedGameObject.name);

if (items[currentID].itemGameObj.GetComponent<Image>().sprite.name == "flaskInv")

{

Debug.Log("Использовано: ЗЕЛЬЕ.");

currentHP += 25;

hp.text = "Current HP: " + currentHP;

UseItem(currentID, data.items[0], 0);

}

else if(items[currentID].itemGameObj.GetComponent<Image>().sprite.name == "keyInv")

{

Debug.Log("Использовано: КЛЮЧ.");

}

else

{

Debug.Log("Ячейка инвентаря пуста.");

}

}

// Метод, который обрабатывает использование предметов

public void UseItem(int id, Item item, int count)

{

items[id].id = item.id;

items[id].count = count;

items[id].itemGameObj.GetComponent<Image>().sprite = item.img;

items[id].itemGameObj.GetComponentInChildren<Text>().text = "";

}

public void MoveObject()

{

Vector3 pos = Input.mousePosition + offset;

pos.z = InventoryMainObject.GetComponent<RectTransform>().position.z;

movingObject.position = cam.ScreenToWorldPoint(pos);

}

public ItemInventory CopyInventoryItem(ItemInventory old)

{

ItemInventory New = new ItemInventory();

New.id = old.id;

New.itemGameObj = old.itemGameObj;

New.count = old.count;

return New;

}

}

[System.Serializable]

// Класс реализующий предмет инвентаря

public class ItemInventory

{

public int id;

public GameObject itemGameObj;

public int count;

}

**Листинг *PlayerController*.*cs*:**

using UnityEngine;

// Класс взаимодействия игрового персонажа на сцене с другими объектами

public class PlayerController : MonoBehaviour

{

public Inventory inv;

public Database data;

public void OnTriggerEnter2D(Collider2D collision)

{

Debug.Log("Произошла коллизия.");

if (collision.tag == "Flask")

{

inv.AddItem(0, data.items[1], 1);

Debug.Log("Подобрано: Зелье.");

Destroy(collision.gameObject);

}

if (collision.tag == "Key")

{

inv.AddItem(0, data.items[2], 1);

Debug.Log("Подобрано: Ключ.");

Destroy(collision.gameObject);

}

}

}