СОДЕРЖАНИЕ

Введение 3

1 Общие сведения об ООО«АйтиТехноСтарт» 4

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

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

2 Обзор среды разработки *Unity* и *Unreal* *Engine* 9

2.1 Среда разработки *Unity*...........................................................................9

2.2 Среда разработки *Unreal Engine*……...................................................12

2.3 Сравнение *Unreal* *Engine* и *Unity* ……………………………………..16

3 Архитектура игрового приложения «2*D*-игра в жанре *SandBox*»…………...18

3.1 Описание задания на преддипломную практику.................................18

3.2 Основные компоненты игрового интерфейса меню...........................18

3.3 Игровые механики.................................................................................19

3.4 Верификация игрового приложения «2*D*-игра в жанре *SandBox*».....24

Заключение 26

Список используемых источников 27

Приложение А Листинг программы 28

# ВВЕДЕНИЕ

Преддипломная практика является важнейшей частью учебного процесса при подготовке высококвалифицированного специалиста. Практика позволяет студентам после изучения ряда теоретических курсов применить все полученные знания на практике, а также закрепить их. В ходе прохождения практики студент получает возможность приобретения опыта работы в коллективе, изучения структуры и организации предприятия, ознакомления со специальным программным обеспечением и оборудованием, применяемым на предприятии.

Задачей данной практической работы является создание игрового приложения «2*D* игра в жанре *SandBox*».

В качестве предприятия для прохождения практики выбрана организация ООО «АйтиТехноСтарт».

Индустрия видеоигр очень сильно развивается в наше время, что позволяет ей быстро и легко предоставить людям игры любого жанра и тем самым удовлетворить предпочтения практически каждого игрока. Вычислительная мощность современных цифровых устройств способна воспроизводить компьютерную графику практически любой сложности, что помогает играм конкурировать с киноиндустрией в качестве изображения.

На данный момент, игры являются одним из наиболее популярных и легкодоступных источников удовольствий. Имеется возможность играть в них как одному, так и в компании нескольких людей, что помогает людям совершенствовать свои социальные навыки. Видеоигры доступны практически на всех современных гаджетах: компьютерах, смартфонах, телевизорах, консолях, электромобилях и даже на электронных часах. В видеоигры играют люди всех возрастов, так как видеоигры помогают провести досуг, и, к тому же, многие из них имеют достаточно низкий порог вхождения.

Подытожив вышесказанное можно сделать вывод, что разработка приложения для ООО «АйтиТехноСтарт» – это хороший опыт для молодого специалиста и разработанное приложение может стать необходимым самому предприятию.

# 1 ОБЩИЕ СВЕДЕНИЯ ОБ ООО «АЙТИТЕХНОСТАРТ»

1. **История организации**

Общество с ограниченной ответственностью «АйтиТехноСтарт» создано в 2016 году. Учредитель – закрытое акционерное общество «Гомельлифт». Основная деятельность – дополнительное образования детей и молодёжи. В октярбре 2016 года учреждение сменило юридический адрес и является арендатором помещения в «Пушкин Плаза». Количество сотрудников – более 15 человек, из них более 10 ­– педагогических работников, имеющих преимущественно 1 и высшую педагогическую категорию.

Количество обучающихся – от 100 до 200 человек, т.к. предприятие появилось относительно недавно. Ежегодно учащиеся центра являются победителями республиканских и международных конкурсов и соревнований. По итогам 2020 года учреждение занесено на республиканскую доску почёта [1].

На базе общества с ограниченной ответственностью реализуется инновационный проект «АйтиТехноСтарт». 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.



Рисунок 1.1 – Логотип учреждения

С 2016 года на базе организации реализуется проект обучения учащихся. Открыты 4 программы обучения: вундеркинды, 1 уровень (6-8 лет), 2 уровень (9-11 лет), 3 уровень (12-15 лет).

Программа вундеркинд включает:

– робототехника *Lego* *Robots* *WEDO* – в процессе изучения курса дети научатся проектировать и собирать рабочие модели, разовьют навыки творческого поиска решений, конструирования;

– интеллектуальная лаборатория, увлекательная математика *Lego* *education* *moremath* – программа данного курса знакомит детей с математическими понятиями при помощи веселых практических занятий. Позволит развить у детей нестандартное мышление, мелкую моторику;

– исследовательская лаборатория: Айти-Мир – данный курс направлен на расширение кругозора ребенка, на развитие умения мыслить и размышлять. Физические и химические опыты в занимательной форме познакомят детей с разнообразными применениями законов физики и химии. Интересные опыты расширят знания детей, будут способствовать развитию логического мышления, привьют интерес к наукам;

– творческая лаборатория: Айти-Арт – в ходе занятий дети научатся создавать удивительные яркие и красивые картинки с помощью компьютера, фантазировать и просто рисовать в свое удовольствие. Занятия не только развивают эстетический вкус и творческое воображение ребенка, но и знакомят с окружающим миром.

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). Награждение дипломов за успешное обучение и прохождение республиканского конкурса рисунок 1.2.



Рисунок 1.2 – Награждение дипломов за успешное обучение и прохождение республиканского конкурса

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 СРЕДЫ РАЗРАБОТКИ *UNITY* и *UNREAL ENGINE*

* 1. Среда разработки *Unity*

*Unity* ­– межплатформенная среда разработки компьютерных игр. *Unity* позволяет создавать приложения, работающие под более чем 20 различными операционными системами, включающими персональные компьютеры, игровые консоли, мобильные устройства, интернет-приложения и другие. Выпуск *Unity* состоялся в 2005 году и с того времени идёт постоянное развитие.

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 [7].

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.

Редактор *Unity* имеет простой *Drag&Drop* интерфейс, который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Движок поддерживает два скриптовых языка: *C*#, *JavaScript* (модификация). Ранее была поддержка *Boo* (диалект *Python*), но его убрали в 5-й версии. Расчёты физики производит физический движок *PhysX* от *NVIDIA*. Среду разработки *Unity*, можно увидеть на рисунке 2.1.

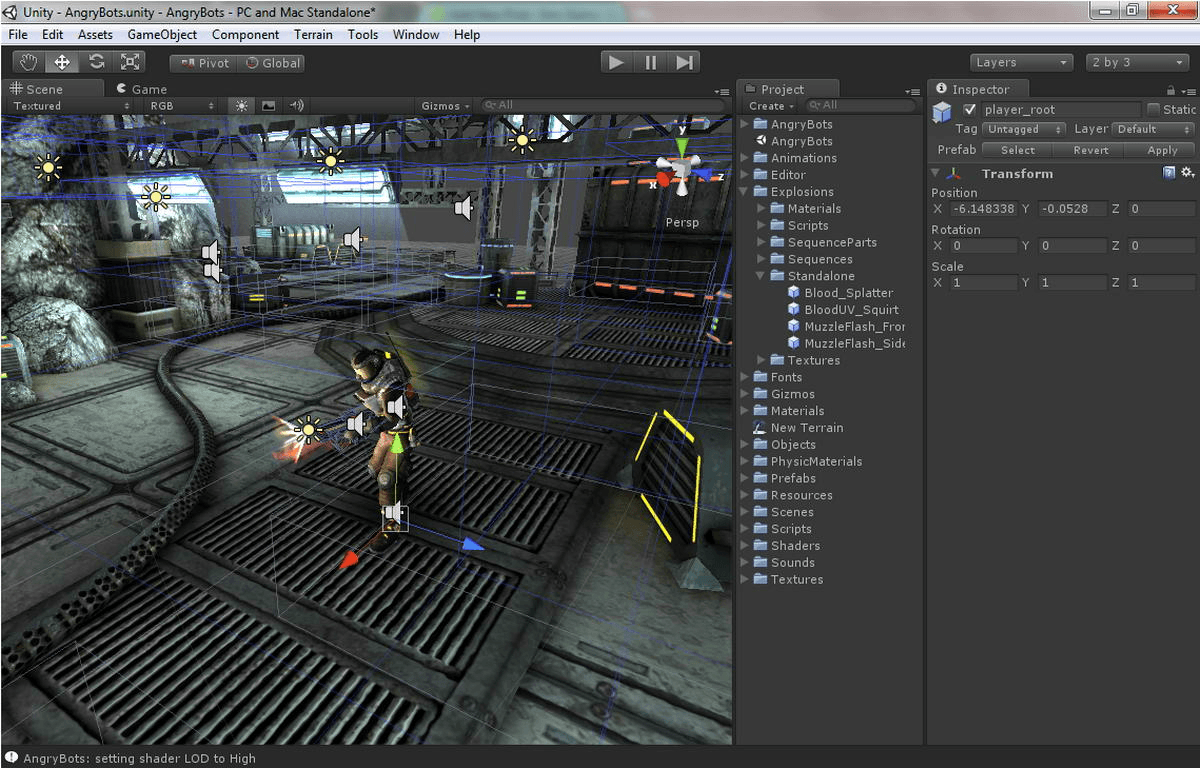


Рисунок 2.1 – Среда разработки *Unity*

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 (в *Unity* допускается наличие двух и более объектов с одинаковыми названиями), 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 [5].

К объектам можно применять коллизии (в *Unity* т. н. коллайдеры – *collider*), 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) [6].

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

Достоинства и недостатки игрового движка:

– как 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. Компьютерные игры на *Unity* охватывают множество платформ и жанров, характерными примерами которых являются:

– *Guns* *of* *Icarus* *Online*, *Gone* *Home* – шутер от первого лица и квест от первого лица, созданные независимыми студиями – для персональных компьютеров;

– *Dead* *Trigger*, *Bad* *Piggies*, *Tyrant* *Unleashed* – шутер от первого лица, головоломка и коллекционная карточная игра – для мобильных устройств;

– *Assault* *Android* *Cactus*, *The* *Golf* *Club* – аркадный шутер и спортивный симулятор – для игровых консолей.

Пример игры на *Unity* можно увидеть на рисунке 2.2.



Рисунок 2.2 – Игра *Guns* *of* *Icarus* *Online* на *Unity*

1. **Среда разработки Unreal Engine**

*Unreal Engine* (*UE*) – это движок для создания игр, один из двух наиболее популярных в мире. Движком называется программная среда, на основе которой конструируются и создаются игры. В ней можно работать с персонажами, логикой, физикой и графикой игры.

*UE* разработала компания *Epic* *Games* для своей игры под названием *Unreal*, и после этого движок стал популярен. Его основное отличие – хорошая оптимизация: *Unreal Engine* создавался не как отдельный коммерческий продукт, а как рабочий инструмент, и ориентирован он на 3*D*-игры.

Наиболее известная сейчас версия – *Unreal Engine* 4, или *UE*4. Но недавно вышел *Unreal Engine* 5, а некоторые игры до сих пор написаны на старых версиях 2 или 3. С каждым обновлением доступная графика становится все более мощной, возможности повышаются и позволяют создавать все более сложные и реалистичные игры.

Название читается как «анрил энджин», иногда его сокращают просто до «анрил».

*Unreal Engine* используется для:

– разработка игра для ПК и консолей. *Unreal* *Engine* традиционно лучше всего подходит для создания трехмерных мощных игр для компьютеров и консолей. Он изначально создавался для этой цели, возможность делать игры в 2*D* и под другие устройства в нем появилась относительно недавно. На *Unreal* *Engine* написаны многие известные игры *AAA*-класса, то есть высокобюджетные, с хорошей графикой и рассчитанные на широкую аудиторию, – игры-блокбастеры. Впрочем, *UE* подходит и для создания низкобюджетных инди-игр, особенно если разработчикам важны графика и оптимизация;

– мобильная игровая разработка. Инструменты для создания мобильных игр в *UE* появились позже. До сих пор считается, что для ПК-игр движок подходит лучше, а для мобильных стоит использовать *Unity*. Но *Unreal Engine* занял свое место и в этой нише. Он используется для создания широкого спектра игр, чаще всего трехмерных. Инструменты для создания 2*D*-игр там появились, начиная с *UE*4 и поддерживаются в новой пятой версии, но среди разработчиков движок все еще считается не лучшим выбором для создания двухмерных игр;

– неигровые варианты применения. *Unreal Engin*e можно использовать не только в игровой индустрии. Его применяют в сфере *VR/AR*-технологий, а также в кинематографе для создания анимаций и компьютерной графики. С помощью *UE* создают ролики для специальных кинотеатров, которые показывают видео с обзором в 360°. Еще движок применяют на телевидении: он позволяет накладывать эффекты на видео, идущее в прямом эфире.

Если говорить об особенностях *Unreal Engine*, можно отметить ориентированность на 3*D*. Изначально движок создавался для внутренних нужд компании *Epic Games*. Она разрабатывала на нем собственные игры, а ее проекты были трехмерными. Поэтому поддержка двумерных проектов была слабой. Но движок оказался таким удачным, что им начали пользоваться и другие игровые разработчики. Тем не менее ориентированность на 3*D*-игры сохранилась, и *Epic Games* начали добавлять больше возможностей для двумерных игр относительно недавно. Также немаловажной частью движка *Unreal Engine* является мощная опитимизация и *С*++. В качестве языка программирования для *Unreal* *Engine* используется *C*++. Это мощный, быстрый, но довольно сложный язык, который непросто изучить с нуля. Тем не менее его применение позволяет хорошо оптимизировать игры. Это важное отличие *UE* от другого популярного движка, *Unity*: создать игру сложнее, но, если получится – она, скорее всего, будет быстрее и эффективнее.

Писать на *C*++ сложно, а с движком работают не только программисты, но и, например, художники-аниматоры. Поэтому *Epic Games* разработали для *UE* внутренний язык визуального программирования, который называется *Blueprints*. Это способ программировать без написания кода – создавать программы из специальных визуальных блоков и связей между ними. Писать так игры легче, и способ подходит даже для тех, кто незнаком с *C*++. Однако сложную логику все же лучше реализовывать с помощью кода. *Blueprints* облегчает задачу, но он не универсален.

На рисунке 2.3 можно увидеть программную среду разработки *Unreal Engine*.

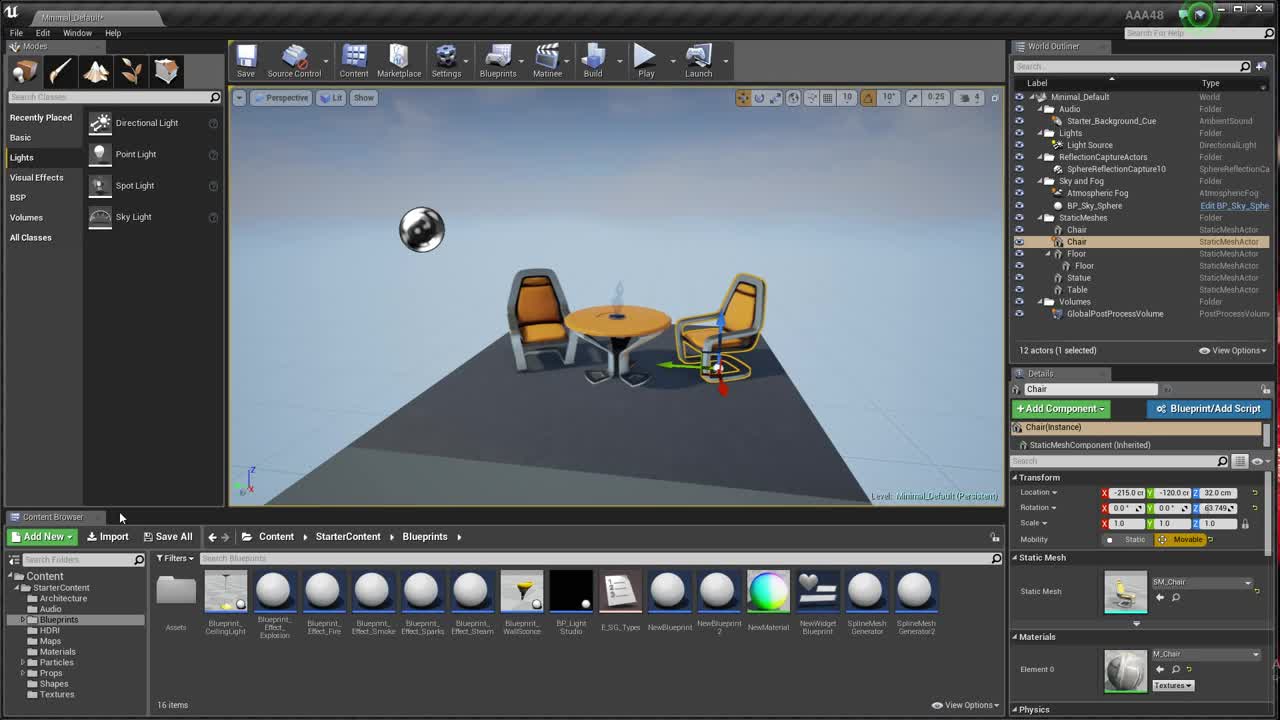


Рисунок 2.3 – Программная среда разработки *Unreal Engine*

В *Unreal Engine* огромное количество возможностей для создания фотореалистичной трехмерной графики. В нем множество текстур, визуальных эффектов и материалов, которые можно применить к объектам, чтобы изменить их внешний вид. Графика гибко настраивается, в результате можно создавать какие угодно материалы, поверхности и эффекты, задавать им различные параметры и смешивать друг с другом.

Эффекты – это не только поверхности и текстуры. Например, *UE* позволяет генерировать мелкие частицы с помощью внутреннего инструмента *Cascade*. С его помощью можно создавать пыль, дождь, снег и многое другое, причем частицы будут реалистично выглядеть и двигаться.

Для анимации предметов и персонажей можно использовать визуальный модуль *Blueprints Animation*. Задачу облегчают готовые паттерны движений для разных видов моделей. Можно взять их и доработать под свои нужды. А если для игры нужно что-то совсем необычное, анимацию можно гибко настроить с нуля рисунок 2.4.

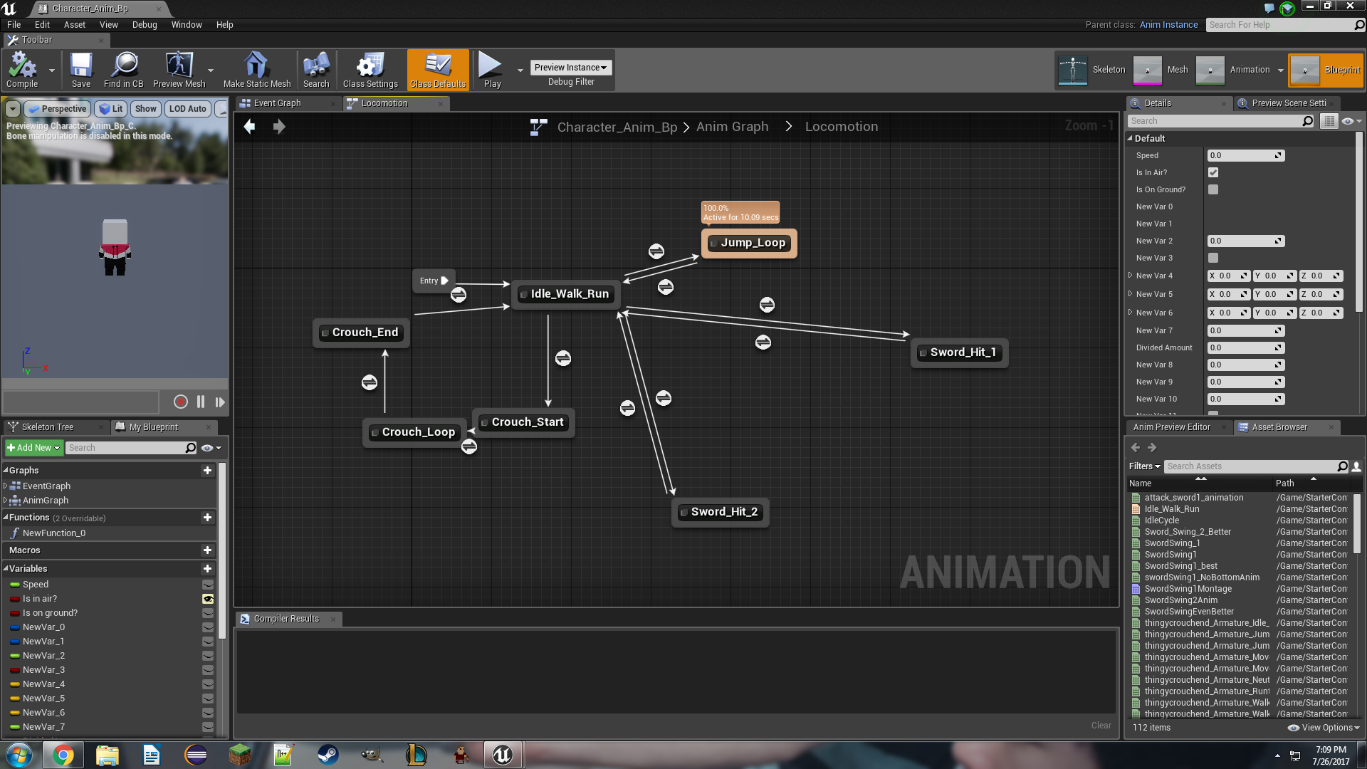


Рисунок 2.4 – Анимации при помощи *Blueprints* *Animation*

Так же гибко в *UE* можно настраивать звук. Встроенные инструменты позволяют переключать аудиофайлы в зависимости от сцены и действий, менять их громкость, смешивать друг с другом, накладывать эффекты и делать многое другое. В движке есть модуль *Sound* *Cue*, который непосредственно занимается звуковыми эффектами. Кстати, в виду имеется не только музыка, но и диалоги, реплики или различные фоновые звуки.

1. **Сравнение Unreal Engine и Unity**

Преимущества *Unity:*

– *Unity* простой и удобный движок. *Unity* легко пользоваться, а его интерфейс зачастую интуитивно понятен.В нем можно создавать 3*D* и 2*D* игры, а так же делать их под *Web*,*Android* и *IOS* лишь не сильно изменив код. Так же в нем есть встроеный редактор кода, но также можно установить дополнение к *Visual Studio* для более удобной разработки на *Unity*. А если есть проблемы при написании кода, то на сайте движка есть хорошая документация к его библиотекам, а также уроки по созданию простых игр и раскрытию способностей движка;

– *Unity* использует *C*#. *C*# – это *C* подобный язык, который вобрал в себя лучшие черты от *C*++ и *Java*. Он легок в изучении, удобен в использовании и легко читаем. Но в отличии от этих языков он не кросплатформенный сам по себе, но благодаря *Unity*, код написанный на языке *C*# легко компилируется под разные платформы и исправно там работает;

– богатый *Asset* *Store*. В *Unity* есть свой магазин с ассетами, это место, где люди выкладывают свои законченные проекты, упрощающие разработку или улучшающие игру бесплатно или за деньги. Например, вы можете для мобильной игры скачать уже готовый джойстик или найти бесплатные 3д модели для человека. Так же некоторые *Unity* разработчики зарабатывают тем, что выкладывают свои наработки за деньги;

– постоянно совершенствуется. На *Unity* регулярно выходят обновления, исправляющие ошибки, добавляющие новые фишки, например, как было с *Unity* 5. Это было глобальное обновление, добавляющее такую полезную функцию, как *UI* в игру. Тогда разработка интерфейса пользователя для игрока стала очень простой, в отличии от предшествующей ей *GUI*, где для отрисовки интерфейса надо было в коде писать его координаты относительно экрана пользователя.

Первая область сравнения – *UI*-редакторы для создания уровней, которые, по мнению автора, очень похожи. В них есть браузеры контента для ассетов, скриптов и других файлов проекта. Игровые объекты можно перетаскивать в область сцены и таким образом добавлять в её иерархию.

Объекты в редакторе сцены изменяются с помощью инструментов перемещения, поворота и масштабирования – они похожи в обоих движках. Свойства *Unity*-объектов отображаются в *Inspector*, а *UE*4 – в части *Details*. *Jayanam* также сравнивает возможности *Unity* *Prefabs* c *Blueprints*.

В обоих движках есть статические меши (*static meshes*) – их можно двигать, поворачивать, и масштабировать – и скелетные меши (*skeletal meshes*) – геометрические объекты, привязанные к костям скелета и используемые для анимирования персонажей. Их можно создавать в программах вроде *Blender* или *Maya*.

*Unity* 5 поддерживает языки *C*# и *UnityScript*. *API* и его концепт очень похож на аналог из *UE*4. При использовании управляемого языка вроде *C*#, программист не обязан использовать указатели (*pointers*), компилирование происходит быстро. В *Unity* нет системы визуального скриптования, и чтобы использовать что-то подобное, разработчик вынужден покупать сторонние дополнения «*Playmaker*».

Для 2*D*-разработки в *Unity* есть великолепные инструменты – *sprite creator*, *sprite editor* и *sprite packer*. *UE*4 также поддерживает спрайты в *Paper* 2*d*, но решения из *Unity* мощнее, кроме того, в последнем есть отдельный физический движок для 2*d*-объектов.

В *UE*4 встроен постпроцессинг. К сцене можно применять *bloom*-эффект, тонирование и антиалиасинг как глобально, так и к отдельным её частям (при помощи компонента *PostProcessVolume*).

В *Unity* есть стек постпроцессинга, который можно скачать из магазина ассетов движка. Система менее гибкая, чем в *UE*4 – эффекты применяются только стеком или скриптами к камере.

Если подвести итоги сравнения, было принято решение реализовывать поставленную задачу на платформе *Unity*, так как на нем проще всего реализовать 2*D* игру, есть большое количество различных 2*D* редакторов, по сравнению с *Unreal Engine*.

# АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «2*D-*ИГРА В ЖАНРЕ *SANDBOX*»

1. **Описание задания на преддипломную практику**

Необходимо разработать игровое приложение «2*D* игра в жанре *SandBox*» для игры пользователя на персональном компьютере. Игровое поле представлено от третьего лица. Цель игрового приложения заключается в ознакомлении с механиками и проработкой мелких деталей.

Для реализации приложения использован игровой движок *Unity* и средства языка программирования *C*#. 2*D*-модели, используемые в игровом приложении, были разработаны и редактированы в *Blender* и *Adobe Photoshop*.

1. **Основные компоненты игрового интерфейса меню**

Первоначально необходимо было создать сцену *Menu*.

В сцене *Menu* созданы все необходимые объекты и компоненты, для реализации игрового меню рисунок 3.1.



Рисунок 3.1 – Главное меню игрового приложения

Игровое меню содержит *Canvas* с текстом и кнопками. Каждая кнопка имеет свою логику, так как она переносит игрока игровое поле. Всего в игре присутствует одна сцена с уровнем. Также в игре присутствует анимация персонажей, сделано это при помощи *Animator* и *Animation* покадрово.

Для логики перехода на уровень по кнопке, был добавлен пустой объект, на который был навешен скрипт *Scenes*, которые имеет метод *PlayGame*. При этом задействуется встроенная в *Unity* библиотека *UnityEngine*.*SceneManagment*.

Далее в кнопках в вкладке *Inspector* есть метод *On* *Click*, в котором нужно вставить этот пустой объект со скриптом, а рядом указать уровень, на который нужно переключиться, уровни можно заметить на рисунке 3.2.

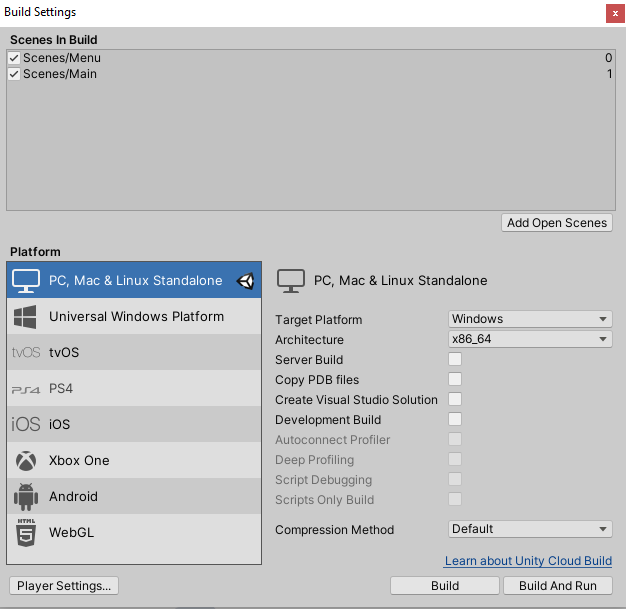


Рисунок 3.2 – Расположение уровней

1. **Игровые механики**

В игре присутствует большое количество игровых объектов и компонентов, к примеру:

– сам игровой персонаж, с анимацией прыжка, смерти, бега, столкновения со стенами;

– враг, который стоит статически, у которого присутствует анимация атаки, передвижения. У врага есть свой коллайдер, благодаря которому, если игрок заходит на его границу, персонаж теряет здоровье;

– задний фон, который двигается с постоянной скоростью и если он заходит за рамки камеры, то он возвращается в начальную позицию;

– инвентарь, если персонаж сломает какой-либо блок, то блок выпадет, у него присутствует анимация и блок имеет гравитацию и если персонаж подбирает блок, он появляется слева сверху в инвентаре;

– сам уровень, которые содержит в себе больше тысячи мелких деталей, которые объединены в огромную карту;

– игровой мир, который представляет собой физически круглую карту, в которой игрок может передвигаться влево, с изменением камеры и вправо соответственно;

– игрок может на правую кнопку мыши ставить блоки, выбирать с помощью левой кнопки мыши в инвентаре их и использовать блоки на игровой сцене.

Одной из главных механик в игре, является типичное для *SandBox* игр передвижение персонажа, вверх, вниз, вправо и влево, при помощи клавиш управления *W*, *A*, *S*, *D*.

Передвижение игрока и вся логика, связанная с ним, содержится в скрипте *CharacterController2D.cs*. Также в этом скрипте описана частично анимация персонажа, поворот его в методе *Flip*, прыжки персонажа, взаимодействие его с объектами и врагами, большое количество различных проверок с поверхностью и врагом. Анимация при получении урона и перезагрузки уровня описана в данном скриптерисунок 3.3.

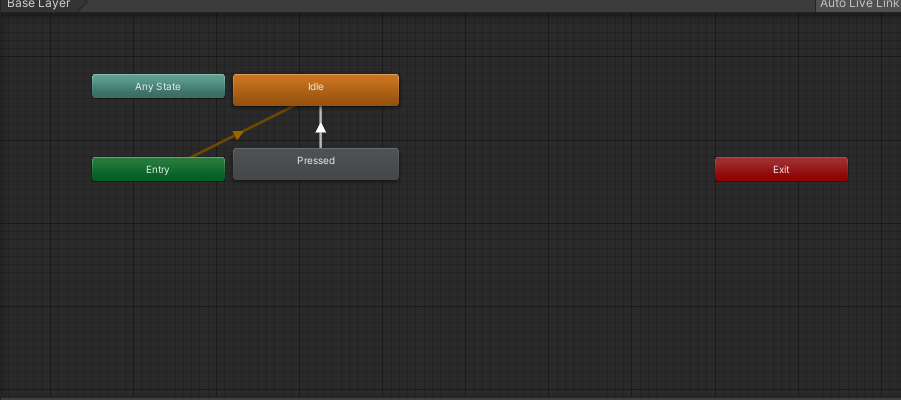


Рисунок 3.3 – *Animator* игрока

Второй из главных механик, связанной с игроком, является то, что, когда он прыгает либо соприкасается со стеной, игрок меняет свою анимацию. Это все описано так же в скрипте *CharacterController2D.cs*.

В скрипте *CharacterAnim Import.cs* реализована смена анимации врага при его различных действиях, как при смерти врага, так и при его атаке. Вся логика врага, описана в скрипте *CharacterLogic*.*cs*, а именно скорость врага, диапазон видимости в близости персонажа, патрулирование местности, поворот врага. В скрипте *HpController.cs*, описана логика получение повреждения персонажа врагом. Так же при взаимодействии врага с каким-либо объектом, происходит характерный звук, а также при взаимодействии с игроком. Аналогично игроку, у врагов есть свои анимации, которые описаны так же в этом скрипте рисунок 3.4

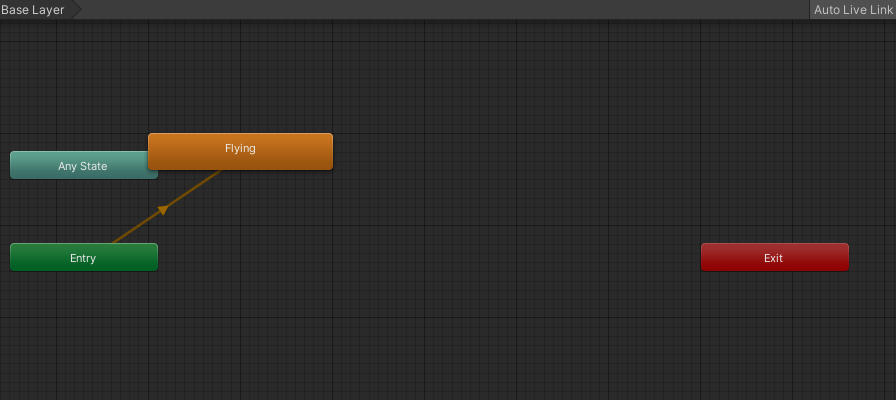


Рисунок 3.4 – *Animator* врага

Следующим немаловажным введением в игру, является инвентарь персонажа, который расположен в левой верхней части экрана. Логика инвентаря у персонажа сделана максимально просто, для приятной картинки для глаз. При разрушении блока на уровне, он падает вниз на землю и после этого игрок может подойти и подобрать этот предмет, предмет исчезает с игрового поля и появляется в игровом инвентаре, для каждого предмета сделана своя ячейка хранения его, так же с этого инвентаря игрок может поставить блок куда ему захочется в любую часть карты, ограничений нет рисунок 3.5.



Рисунок 3.5 – Реализация инвентаря

Для реализации игрового мира, с помощью создания из блоков, необходимо для начала было сделать скрипты *Ground.cs* и *Tree.cs*. В скрипте *Ground.cs* реализована механика сращивания объектов с друг другом, путем проверок есть ли у блока конкретная текстура, если ее нет, то меняется на другую и т.д. Когда персонаж ломает какой-либо блок, в основном это земля и, если он захочет в дальнейшем поставить его, спрайт поставленного блока и спрайт блоков, которые генерируются в мире будут отличаться, поэтому необходимо добавить данную механику. Деревья реализованы в скрипте *Tree.cs*, у каждого блока есть своя прочность, с мощью которой, персонаж может добыть его левой кнопкой мыши, в данном скрипте реализована логика дерева, его форма как оно будет генерироваться, при разрушении менять размер и текстуру и т.д. рисунок 3.6.

**

Рисунок 3.6 – Персонаж, земля и деревья

Конкретно генерация мира описана в скрипте *GroundGenerator*.*cs* в котором содержится вся необходимая информация о текущем состояния мира, количества блоков, содержащихся в нем, размере его. Блоки генерируются случайно, но со своей логикой. Блоки земли, с текстурами травы генерируются на верхней части игрового мира, деревья генерируются над блоками, с используемыми проверками, задний фон блоков, генерируется у каждого блока по-своему, руда, камень, земля генерируются случайно рисунок 3.7.



Рисунок 3.7 – Генерация мира

В проекте присутствуют три интерфейса, который отвечают за различные функции:

– *IEntity* – интерфейс врага;

– *IObject* – интерфейс, который отвечает за состояние блока, проверяет что находится как физический предмет, или как материальный;

– *ITool –* интерфейс описывающий инвентарь, проверка может ли персонаж нанести урон какому-либо врагу, кол-во секунд, затраченное на копание одного блока.

Выпадение блока после вскапывания персонажа, реализована в скрипте *DropCollider.cs*. В скрипте реализован список, который принимает блоки, содержащиеся на карте, а также список объектов, которые удалены с игрового поля, путем подбора блока персонажем. Каждый блок имеет свой коллайдер, благодаря которому, при взаимодействии с персонажем, блок попадает в инветарь.

Большую часть игровой сцены занимают сами стены, которые статически, то есть неподвижно стоят и блокирует продвижение игрока. Это было сделано для того, чтоб игрок не выпадал за уровень и для удобства самой игры.

1. **Верификация игрового приложения «2*D* игра в жанре *SandBox*»**

На главном и единственном уровне игры, игрок может заметить перед собой огромный мир, который разбит на несколько частей:

– деревья с листвой;

– блоки;

– блоки с травой, которая зависит от расположения блока;

– враг в виде скелета;

– сам игровой персонаж, с присущими ему анимациями.

На рисунке 3.8 можно увидеть персонажа в анимации.



Рисунок 3.8 – Игровой персонаж с анимацией прыжка

Игрок может передвигаться при помощи клавиш *W*, *A*, *S*, *D*. На уровне можно встретить врага, который патрулирует определенную местность, но у него стоят ограничения, он может патрулировать, но не может упасть ниже той поверхности, на которой находится в данный момент. Мир собой представляет 200x50 блоков, которые генерируются абсолютно случайно. Игрок может кроме того, что передвигаться по игровому полю, так же совершать прыжки, строить блоки при помощи правой кнопки мыши и ломать их при помощи левой кнопки мыши. У игрока есть инвентарь, в случае если персонаж вскопает какой-либо блок, он пропадает и появляется маленькая версия этого блок, которую уже в дальнейшем может подобрать персонаж.

На рисунке 3.9 можно увидеть врага, который замахивается перед ударом.



Рисунок 3.9 – Враг, который атакует персонажа

Все текстуры и спрайты были сделаны в программной среде *Blender* и *Photoshop*. В ходе продолжительного времени получилось сделать множество различных интересных механик и разработано игровое приложение «2D игра в жанре *SandBox*» при использовании игрового движка *Unity* и средств языка программирования *C*#.

# ЗАКЛЮЧЕНИЕ

Прохождение преддипломной практики является важным элементом учебного процесса по подготовки специалиста в области программирования.

Во время её прохождения будущий программист применяет полученные в процессе обучения знания, умения и навыки на практике.

Основными задачами преддипломной практики являются:

* получения практического опыта работы в качестве программиста;
* улучшения качества профессиональной подготовки;
* закрепление полученных знаний по общим и специальным дисциплинам.

Широкий охват отраслей, с которыми пришлось сталкиваться на практике, позволил лучше усвоить изученный теоретический материал, полученный на занятиях в колледже.

Были освоены некоторые тонкости применения ПО на практике, была изучена работа некоторых программ, подпрограммы, которые были необходимы при разработке индивидуального задания.

Практическая деятельность помогла научиться самостоятельно решать определенный круг задач, возникающих в ходе работы программиста.

В процессе прохождения преддипломной практики было разработано игровое приложение «2*D* игра в жанре *SandBox*», схожее с реальными задачами клиентов, поставленными перед компанией.

# Список используемых источников

1. Официальный сайт ООО «Айтилайт» [Электронный ресурс]. Режим доступа: [*https://it-light.by/news?start=16*](https://it-light.by/news?start=16%20)  – Дата доступа: 04.04.2023.
2. Юридический словарь [Электронный ресурс]. Режим доступа: *http://multilang.pravo.by/ru/Term/Index/525?langName=ru&ch=Все&size=25&page=4&type=3* – Дата доступа: 04.07.2022.
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 *C*#/ Дж.Хокинг. – Питер, 2019. – 352 с.
5. Основы разработки 2D-игр [Электронный ресурс]. Режим доступа: *https://proglib.io/p/sozdaem-2d-igru-na-unity-instrukciya-dlya-novichka-2020-09-01* – Дата доступа: 04.07.2022.
6. Руководство: 2D или 3D проекты [Электронный ресурс]. Режим доступа: *https://docs.unity3d.com/ru/530/Manual/2Dor3D.html* – Дата доступа: 04.07.2022.
7. Unity, платформа разработки в реальном времени [Электронный ресурс]. Режим доступа: *https://unity.com/ru* – Дата доступа: 04.07.2022.

# ПРИЛОЖЕНИЕ А

(обязательное)

# Листинг программы

**Листинг Block.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Block : MonoBehaviour, IObject

{

public string name = "block";

public float DropUnits = 0;

public int plane = 0;

public Sprite DropSprite;

public AudioClip drilSound;

public AudioClip dropSound;

private int phisicsLayer = 1;

private int renderLayer = 3;

public List<Condition> conditions { get; private set; }

public Condition CurrentCondition { get; private set; }

private bool isPressed = false;

private float timeStart;

protected GrounGenerator grounGenerator;

public AudioSource audioSource;

private BoxCollider2D boxCollider2D;

private Rigidbody2D rigidbody2D;

protected SpriteRenderer spriteRenderer;

public Block()

{

CurrentCondition = Condition.Material;

}

protected virtual void OnMouseDown()

{

isPressed = true;

timeStart = Time.time;

audioSource.clip = drilSound;

}

protected virtual void OnMouseExit()

{

isPressed = false;

}

protected virtual void OnMouseUp()

{

isPressed = false;

}

protected virtual void Start()

{

switch(plane)

{

case 0: renderLayer = 3; phisicsLayer = 1; transform.position += new Vector3(0, 0, -0.1f); break;

case 1: renderLayer = 2; phisicsLayer = 2; transform.position += new Vector3(0, 0, -0.05f); break;

case 2: renderLayer = 1;

GetComponent<SpriteRenderer>().color -= new Color(0.3f, 0.3f, 0.3f, 0);

Destroy(GetComponent<BoxCollider2D>());

Destroy(GetComponent<Rigidbody2D>());

break;

}

spriteRenderer = GetComponent<SpriteRenderer>();

rigidbody2D = GetComponent<Rigidbody2D>();

grounGenerator = transform.parent.GetComponent<GrounGenerator>();

boxCollider2D = GetComponent<BoxCollider2D>();

spriteRenderer.sprite = DropSprite;

grounGenerator.SetElementAtPosition(plane, (int)transform.position.x, (int)transform.position.y);

gameObject.layer = LayerMask.NameToLayer("Layer"+ phisicsLayer);

GetComponent<SpriteRenderer>().sortingLayerName = "Layer" + renderLayer;

}

protected virtual void Update()

{

if(CurrentCondition == Condition.Material)

if (isPressed)

{

if (!audioSource.isPlaying)

audioSource.Play();

if (Time.time - timeStart >= DropUnits) Drop();

}

}

protected virtual void Drop()

{

spriteRenderer.sprite = DropSprite;

rigidbody2D.isKinematic = false;

boxCollider2D.enabled = true;

gameObject.layer = LayerMask.NameToLayer("Layer2");

transform.localScale = transform.localScale \* 0.5f;

CurrentCondition = Condition.Dropped;

grounGenerator.DelElementAtPosition(plane, (int)transform.position.x, (int)transform.position.y);

}

public IEnumerator EnterPlayer(Transform playerTransform)

{

rigidbody2D.isKinematic = true;

boxCollider2D.enabled = false;

while ((playerTransform.position - transform.position).magnitude > 0.2f)

{

transform.position = Vector3.MoveTowards(transform.position, playerTransform.position, 10f \* Time.deltaTime);

yield return null;

}

spriteRenderer.enabled = false;

while (audioSource.isPlaying)

yield return null;

audioSource.clip = dropSound;

audioSource.Play();

while (audioSource.isPlaying)

yield return null;

//Destroy(gameObject);

}

public virtual void Instanse(int x, int y)

{

CurrentCondition = Condition.Material;

transform.position = new Vector2(x, y);

rigidbody2D.velocity = new Vector2();

spriteRenderer.enabled = true;

spriteRenderer.sprite = DropSprite;

gameObject.layer = LayerMask.NameToLayer("Layer" + phisicsLayer);

transform.localScale = transform.localScale \* 2f;

boxCollider2D.enabled = true;

grounGenerator.SetElementAtPosition(plane, x, y);

}

}

**Листинг Depend.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Depend : MonoBehaviour

{

public Transform dependObject;

public bool x, y, z;

private void Update()

{

Vector3 new\_pos = dependObject.transform.position;

if (!z) new\_pos.z = 0;

if (!x) new\_pos.x = 0;

if (!y) new\_pos.y = 0;

transform.position = new\_pos;

}

}

**Листинг DropCollider.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.Linq;

using System;

public delegate void ToolBarDelegate(Dictionary<Block, int> toolBarBlocks);

public class DropCollider : MonoBehaviour

{

public GrounGenerator grounGenerator;

public static event ToolBarDelegate UpdateToolBar;

public List<GameObject> myObjects = new List<GameObject>();

static Dictionary<Block, int> toolBarBlocks = new Dictionary<Block, int>();

static List<GameObject> dropZoneObjects = new List<GameObject>();

private void OnGUI()

{

if (Input.GetMouseButtonDown(1))

{

Vector2 position = Camera.main.ScreenToWorldPoint(Input.mousePosition);

int x = Convert.ToInt32(position.x);

int y = Convert.ToInt32(position.y);

int plane = ToolBarController.CurrentType.plane;

// если нет зарегистрированного элемента в данной позиции

if (!grounGenerator.IsElementAtPosition(plane, x, y))

{

// берем текущий экземпляр

GameObject a = ToolBarController.CurrentType.gameObject;

if (a != null && myObjects.Contains(a))

{

a.GetComponent<IObject>().Instanse(x, y);

myObjects.Remove(a);

updateToolBar();

}

}

}

}

private void Update()

{

// список на удаление

var toAdd = dropZoneObjects.Where(obj => obj.GetComponent<IObject>().CurrentCondition == Condition.Dropped).ToList();

if (toAdd.Count > 0)

{

// оставшиеся

dropZoneObjects = dropZoneObjects.Except(toAdd).ToList();

myObjects.AddRange(toAdd);

updateToolBar();

// те, кто в списке на удаление - удаляются

foreach (GameObject obj in toAdd)

StartCoroutine(obj.GetComponent<IObject>().EnterPlayer(transform));

}

}

private void OnTriggerEnter2D(Collider2D collision)

{

IObject obj = collision.GetComponent<IObject>();

if (obj != null)

if (obj.CurrentCondition == Condition.Dropped)

{

StartCoroutine(obj.EnterPlayer(transform));

myObjects.Add(collision.gameObject);

updateToolBar();

}

else

dropZoneObjects.Add(collision.gameObject);

}

private void OnTriggerExit2D(Collider2D collision)

{

IObject obj = collision.GetComponent<IObject>();

if (obj != null)

dropZoneObjects.Remove(collision.gameObject);

}

// обновление для toolBar кидаем новый словарь эл-нт: кол-во

private void updateToolBar()

{

toolBarBlocks = new Dictionary<Block, int>();

Dictionary<string, int> namesDict = new Dictionary<string, int>();

foreach (GameObject obj in myObjects)

{

if (obj.GetComponent<Block>())

{

if (namesDict.ContainsKey(obj.GetComponent<Block>().name)) namesDict[obj.GetComponent<Block>().name] += 1;

else namesDict.Add(obj.GetComponent<Block>().name, 1);

}

}

foreach (var item in namesDict)

foreach (GameObject obj in myObjects)

{

if (obj.GetComponent<Block>())

{

if (obj.GetComponent<Block>().name == item.Key)

{

toolBarBlocks.Add(obj.GetComponent<Block>(), item.Value);

break;

}

}

}

if (UpdateToolBar!= null) UpdateToolBar(toolBarBlocks);

}

}

**Листинг GroundGenerator.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class GrounGenerator : MonoBehaviour

{

[SerializeField] private GameObject groundPref;

[SerializeField] private GameObject stonePref;

[SerializeField] private GameObject freePref;

[SerializeField] private GameObject daimPref;

[SerializeField] private GameObject player;

[SerializeField] private Tree[] trees;

private AudioSource audioSource;

public bool[,,] matrix = new bool[3, 2 \* 1000 , 2 \* 1000 ];

private int matrixOffset;

void Start()

{

audioSource = GetComponent<AudioSource>();

GenerateGround();

}

private void GenerateGround()

{

matrixOffset = matrix.GetLength(1) / 2;

int[] heights = new int[500];

int \_offset = 200;

int cur\_x = -200;

int cur\_y = 0;

while (cur\_x <= 200)

{

int len = Random.Range(7, 30);

int h = Random.Range(-15, 15);

float k = (float)h / len;

// сначала генерируем задний план, потом передний

foreach (int plane in new int[] { 2, 0 })

{

for (int x = cur\_x; x < cur\_x + len; x++)

{

float y = cur\_y + k \* (x - cur\_x);

if (y % 2 != 0) y += 1;

for (int j = -80; j <= y; j++)

{

RandGenerate(plane, new Vector3(x, j));

}

heights[x + \_offset] = (int)y;

}

}

cur\_x += len;

cur\_y += h;

}

// ставим игрока чуть выше верхнего блока

var pp = player.transform.position;

player.transform.position = new Vector3(pp.y, heights[(int)pp.x + \_offset] + 1);

// Создание деревьев

for (int k = -200; k <= 200; k += 2)

{

// с вероятностью 10% создаем дерево

if (Random.RandomRange(0, 101) > 90)

{

Tree cur\_tree = trees[Random.Range(0, trees.Length)];

GameObject[,] blocks = cur\_tree.GetTree();

for (int i = 0; i < blocks.GetLength(0); i++)

for (int j = 0; j < blocks.GetLength(1); j++)

{

if (blocks[i, j] != null)

{

GameObject b = Instantiate(

blocks[i, j],

transform.position + new Vector3(k + j, heights[k + \_offset] + i),

Quaternion.identity);

b.transform.SetParent(transform);

b.GetComponent<Block>().audioSource = audioSource;

}

}

k += blocks.GetLength(0);

}

}

}

private Vector3[] Bezie(Vector3[] p)

{

List<Vector3> a = new List<Vector3>();

for (float t = 0; t < 1; t += 0.1f)

{

float x = (1 - t) \* (1 - t) \* p[0].x + 2 \* (1 - t) \* t \* p[1].x + t \* t \* p[2].x;

float y = (1 - t) \* (1 - t) \* p[0].y + 2 \* (1 - t) \* t \* p[1].y + t \* t \* p[2].y;

a.Add(new Vector3(x,y));

}

return a.ToArray();

}

private void RandGenerate(int plane, Vector3 position)

{

GameObject a = null;

int rnd = Random.Range(0, 1000);

if (rnd >= 0 && rnd < 800) a = groundPref;

else

if (rnd > 800 && rnd < 995) a = stonePref;

else

if (position.y < -15)

a = daimPref;

else

a = stonePref;

if (a != null)

{

GameObject obj = Instantiate(a, transform.position + position, Quaternion.identity);

obj.transform.SetParent(transform);

obj.GetComponent<Block>().plane = plane;

obj.GetComponent<Block>().audioSource = audioSource;

}

}

public bool IsElementAtPosition(int plane, int i, int j)

{

return matrix[plane, i + matrixOffset, j + matrixOffset];

}

public void SetElementAtPosition(int plane, int i, int j)

{

//Debug.Log(plane + " " + (i + offset) + " " + (j + offset));

matrix[plane, i + matrixOffset, j + matrixOffset] = true;

}

public void DelElementAtPosition(int plane, int i, int j)

{

matrix[plane, i + matrixOffset, j + matrixOffset] = false;

}

}

**Листинг ToolBarController.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine.UI;

using UnityEngine;

using System.Collections.Generic;

using System.Linq;

public class ToolBarController : MonoBehaviour

{

public static Block CurrentType = null; // текущий выбранный тип

static int cur\_num\_btn = 0; // текущая выбранная внопка

public Button[] ToolBarButtons; // кнопки

private Dictionary<Block, int> BarBlocks; // текущий набор эл-тов

private void Start()

{

DropCollider.UpdateToolBar += DropCollider\_UpdateToolBar;

ToolBarButtons = new Button[transform.childCount];

for (int i = 0; i < transform.childCount; i++)

ToolBarButtons[i] = transform.GetChild(i).GetComponent<Button>();

}

// обновление списка элементов

private void DropCollider\_UpdateToolBar(Dictionary<Block, int> toolBarBlocks)

{

BarBlocks = toolBarBlocks;

OffButtons();

int i = 0;

foreach (KeyValuePair<Block, int> keyValue in toolBarBlocks)

{

ToolBarButtons[i].transform.GetChild(0).GetComponent<Image>().sprite = keyValue.Key.DropSprite;

ToolBarButtons[i].transform.GetChild(1).GetComponent<Text>().text = keyValue.Value + "";

ToolBarButtons[i].gameObject.SetActive(true);

i++;

if (i == ToolBarButtons.Length) break;

}

UpdateCurType(cur\_num\_btn);

}

// выелючить все кнопки

private void OffButtons()

{

foreach (Button button in ToolBarButtons)

button.gameObject.SetActive(false);

}

// снять выделение со всех кнопок, кроме вызвавшей под номером num

public void DisableToggles(int num)

{

for (int i = 0; i < ToolBarButtons.Length; i++)

{

if (i == num)

ToolBarButtons[i].GetComponent<Outline>().enabled = true;

else

ToolBarButtons[i].GetComponent<Outline>().enabled = false;

}

UpdateCurType(num);

}

// обновление текущего выбранного экземпляра

void UpdateCurType(int num)

{

int j = 0;

foreach (KeyValuePair<Block, int> keyValue in BarBlocks)

{

if (j == num) { CurrentType = keyValue.Key; break; }

j++;

}

cur\_num\_btn = num;

}

}

**Листинг Ground.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Ground : Block

{

public Vector2 RangeOfGrassCount = new Vector2(7,10);

public Sprite SpriteUp;

public Sprite SpriteLeft;

public Sprite SpriteUpLeft;

public Sprite SpriteRight;

public Sprite SpriteUpRight;

public Sprite SpriteMiddle;

public Sprite SpriteUpLeftRight;

public Sprite SpriteLeftRight;

public Sprite SpriteDown;

public Sprite SpriteDownLeft;

public Sprite SpriteDownRight;

public Sprite SpriteDownLeftRight;

public Sprite SpriteDownLeftUp;

public Sprite SpriteDownRightUp;

public Sprite SpriteDownUp;

public Sprite SpriteLeftRightDownUp;

public bool up;

public bool left;

public bool right;

public bool down;

protected override void Start()

{

base.Start();

StartCoroutine(controlGrass());

}

IEnumerator controlGrass()

{

while (true)

{

yield return new WaitForSeconds(Random.Range(RangeOfGrassCount.x, RangeOfGrassCount.y));

if (CurrentCondition == Condition.Dropped) break;

int x = (int)transform.position.x;

int y = (int)transform.position.y;

up = grounGenerator.IsElementAtPosition(plane, x, y +1);

left = grounGenerator.IsElementAtPosition(plane, x - 1, y);

right = grounGenerator.IsElementAtPosition(plane, x + 1, y);

down = grounGenerator.IsElementAtPosition(plane, x, y - 1);

if (up)

{

if (down)

{

if(!left && !right) { spriteRenderer.sprite = SpriteLeftRight; continue; }

if (!right) { spriteRenderer.sprite = SpriteRight; continue; }

if (!left) { spriteRenderer.sprite = SpriteLeft; continue; }

if (left && right) { spriteRenderer.sprite = SpriteMiddle; continue; }

}

else

{

if (!left && !right){ spriteRenderer.sprite = SpriteDownLeftRight; continue; }

if (!right) { spriteRenderer.sprite = SpriteDownRight; continue; }

if (!left) { spriteRenderer.sprite = SpriteDownLeft; continue; }

if (left && right) { spriteRenderer.sprite = SpriteDown; continue; }

}

}

else

{

if (down)

{

if (!left && !right) { spriteRenderer.sprite = SpriteUpLeftRight; continue; }

if (!right) { spriteRenderer.sprite = SpriteUpRight; continue; }

if (!left) { spriteRenderer.sprite = SpriteUpLeft; continue; }

if (left && right) { spriteRenderer.sprite = SpriteUp; continue; }

}

else

{

if (!left && !right) { spriteRenderer.sprite = SpriteLeftRightDownUp; continue; }

if (!right) { spriteRenderer.sprite = SpriteDownRightUp; continue; }

if (!left) { spriteRenderer.sprite = SpriteDownLeftUp; continue; }

if (left && right) { spriteRenderer.sprite = SpriteDownUp; continue; }

}

}

}

}

public override void Instanse(int x, int y)

{

base.Instanse(x, y);

StartCoroutine(controlGrass());

}

}

**Листинг Tree.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[CreateAssetMenu(fileName = "Tree", menuName = "Objects/Tree")]

public class Tree : ScriptableObject

{

public AudioClip woodSound;

public AudioClip dropSound;

public AudioClip foliageSound;

public Sprite Base;

public Sprite Thrunk;

public Sprite Crown;

public GameObject BasePref;

public GameObject ThrunkPref;

public GameObject CrownPref;

public Block GetTree(int i, int j, Block b)

{

Sprite[,] sprites = new Sprite[7, 5]{

{null,Base, Thrunk, null,null},

{null,null, Thrunk, null,null},

{null,null, Thrunk, null,null},

{null,Crown, Thrunk, Crown,null},

{Crown,Crown, Crown, Crown,Crown},

{null, Crown, Crown, Crown, null},

{null, null, Crown, null, null}

};

if (sprites[i, j] != null)

return SettingWood(sprites[i, j], b);

else

return null;

}

public GameObject[,] GetTree()

{

return new GameObject[7,5]{

{null, BasePref, ThrunkPref, null, null},

{null, null, ThrunkPref, null, null},

{null, null, ThrunkPref, null, null},

{null, CrownPref, ThrunkPref, CrownPref, null},

{CrownPref, CrownPref, CrownPref, CrownPref, CrownPref},

{null, CrownPref, CrownPref, CrownPref, null},

{null, null, CrownPref, null, null}

};

}

public Block SettingWood(Sprite sprite, Block block)

{

if (sprite == Crown)

{

block.DropUnits = 0;

block.name = "foliage";

block.drilSound = foliageSound;

}

else

{

if (sprite == Base) block.name = "tree\_base";

else block.name = "tree";

block.drilSound = woodSound;

block.DropUnits = 1;

}

block.dropSound = dropSound;

block.plane = 1;

block.DropSprite = sprite;

return block;

}

public Block SettingWood(Sprite sprite)

{

return new Block { DropUnits=1, name="tree", drilSound=woodSound,

dropSound = dropSound, plane = 1, DropSprite = sprite};

}

}

**Листинг IEntity.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Это интерфейс врага

/// </summary>

public interface IEntity

{

/// <summary>

/// Нанести урон

/// </summary>

void Damage(float damage);

}

**Листинг IObject.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Состояние, может быть в виде физического блока или как дропнутый блок

/// </summary>

public enum Condition {Material, Dropped}

/// <summary>

/// Объект (блоки), который можно собрать

/// </summary>

public interface IObject

{

/// <summary>

/// ВОзможные состояния

/// </summary>

List<Condition> conditions { get; }

/// <summary>

/// Текущее состояние

/// </summary>

Condition CurrentCondition { get; }

IEnumerator EnterPlayer(Transform playerTransform);

void Instanse(int x, int y);

}

**Листинг ITool.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Интрефейс инструмент

/// </summary>

public interface ITool : IObject

{

/// <summary>

/// Нанести урон, возвращает урон, который может быть нанесен сущности

/// </summary>

/// <param name="entity">Сущность, по которой наносят урон</param>

/// <returns></returns>

float GetDamage(IEntity entity);

/// <summary>

/// Копать, возращает кол-во секунд, которое затратит на копание

/// </summary>

/// <param name="obj">Объект, который пытаемся копать</param>

/// <returns></returns>

float GetDig(IObject obj);

}

**Листинг CharacterController2D.cs:**

using UnityEngine;

using UnityEngine.Events;

public class CharacterController2D : MonoBehaviour

{

[SerializeField] private float m\_JumpForce = 400f; // Amount of force added when the player jumps.

[Range(0, 1)] [SerializeField] private float m\_CrouchSpeed = .36f; // Amount of maxSpeed applied to crouching movement. 1 = 100%

[Range(0, .3f)] [SerializeField] private float m\_MovementSmoothing = .05f; // How much to smooth out the movement

[SerializeField] private bool m\_AirControl = false; // Whether or not a player can steer while jumping;

[SerializeField] private LayerMask m\_WhatIsGround; // A mask determining what is ground to the character

[SerializeField] private Transform m\_GroundCheck; // A position marking where to check if the player is grounded.

[SerializeField] private Transform m\_CeilingCheck; // A position marking where to check for ceilings

[SerializeField] private Collider2D m\_CrouchDisableCollider; // A collider that will be disabled when crouching

const float k\_GroundedRadius = .2f; // Radius of the overlap circle to determine if grounded

private bool m\_Grounded; // Whether or not the player is grounded.

const float k\_CeilingRadius = .2f; // Radius of the overlap circle to determine if the player can stand up

private Rigidbody2D m\_Rigidbody2D;

private bool m\_FacingRight = true; // For determining which way the player is currently facing.

private Vector3 m\_Velocity = Vector3.zero;

[Header("Events")]

[Space]

public UnityEvent OnLandEvent;

[System.Serializable]

public class BoolEvent : UnityEvent<bool> { }

public BoolEvent OnCrouchEvent;

private bool m\_wasCrouching = false;

private void Awake()

{

m\_Rigidbody2D = GetComponent<Rigidbody2D>();

if (OnLandEvent == null)

OnLandEvent = new UnityEvent();

if (OnCrouchEvent == null)

OnCrouchEvent = new BoolEvent();

}

private void FixedUpdate()

{

bool wasGrounded = m\_Grounded;

m\_Grounded = false;

// The player is grounded if a circlecast to the groundcheck position hits anything designated as ground

// This can be done using layers instead but Sample Assets will not overwrite your project settings.

Collider2D[] colliders = Physics2D.OverlapCircleAll(m\_GroundCheck.position, k\_GroundedRadius, m\_WhatIsGround);

for (int i = 0; i < colliders.Length; i++)

{

if (colliders[i].gameObject != gameObject)

{

m\_Grounded = true;

if (!wasGrounded)

OnLandEvent.Invoke();

}

}

}

public void Move(float move, bool crouch, bool jump)

{

// If crouching, check to see if the character can stand up

if (!crouch)

{

// If the character has a ceiling preventing them from standing up, keep them crouching

if (Physics2D.OverlapCircle(m\_CeilingCheck.position, k\_CeilingRadius, m\_WhatIsGround))

{

crouch = true;

}

}

//only control the player if grounded or airControl is turned on

if (m\_Grounded || m\_AirControl)

{

// If crouching

if (crouch)

{

if (!m\_wasCrouching)

{

m\_wasCrouching = true;

OnCrouchEvent.Invoke(true);

}

// Reduce the speed by the crouchSpeed multiplier

move \*= m\_CrouchSpeed;

// Disable one of the colliders when crouching

if (m\_CrouchDisableCollider != null)

m\_CrouchDisableCollider.enabled = false;

}

else

{

// Enable the collider when not crouching

if (m\_CrouchDisableCollider != null)

m\_CrouchDisableCollider.enabled = true;

if (m\_wasCrouching)

{

m\_wasCrouching = false;

OnCrouchEvent.Invoke(false);

}

}

// Move the character by finding the target velocity

Vector3 targetVelocity = new Vector2(move \* 10f, m\_Rigidbody2D.velocity.y);

// And then smoothing it out and applying it to the character

m\_Rigidbody2D.velocity = Vector3.SmoothDamp(m\_Rigidbody2D.velocity, targetVelocity, ref m\_Velocity, m\_MovementSmoothing);

// If the input is moving the player right and the player is facing left...

if (move > 0 && !m\_FacingRight)

{

// ... flip the player.

Flip();

}

// Otherwise if the input is moving the player left and the player is facing right...

else if (move < 0 && m\_FacingRight)

{

// ... flip the player.

Flip();

}

}

// If the player should jump...

if (m\_Grounded && jump)

{

// Add a vertical force to the player.

m\_Grounded = false;

m\_Rigidbody2D.AddForce(new Vector2(0f, m\_JumpForce));

}

}

private void Flip()

{

// Switch the way the player is labelled as facing.

m\_FacingRight = !m\_FacingRight;

// Multiply the player's x local scale by -1.

Vector3 theScale = transform.localScale;

theScale.x \*= -1;

transform.localScale = theScale;

}

}

**Листинг PlayerMovement.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public delegate void Void(float a);

public class PlayerMovement : MonoBehaviour, IEntity {

public float Hp = 100;

private float fullHp;

public event Void UpdateHp;

public CharacterController2D controller;

public Animator animator;

public float runSpeed = 40f;

float horizontalMove = 0f;

bool jump = false;

private void Start()

{

fullHp = Hp;

}

void Update () {

horizontalMove = Input.GetAxisRaw("Horizontal") \* runSpeed;

animator.SetFloat("Speed", Mathf.Abs(horizontalMove));

if (Input.GetButtonDown("Jump")&& !jump)

{

jump = true;

animator.SetBool("IsJamping", true);

}

}

public void OnLanding() {

animator.SetBool("IsJamping", false);

}

void FixedUpdate ()

{

// Move our character

controller.Move(horizontalMove \* Time.fixedDeltaTime, false, jump);

jump = false;

}

public void Damage(float damage)

{

Hp -= damage;

UpdateHp(Hp / fullHp);

}

}

**Листинг CharacterAnim.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CharacterAnim : MonoBehaviour

{

public static Animator anim;

void Start()

{

anim = GetComponent<Animator>();

}

void Update()

{

switch (CharacterLogic.CurrentAnimation)

{

case CharacterLogic.allAnimations.Staying:

anim.SetBool("IsRunning", false);

break;

case CharacterLogic.allAnimations.Running:

anim.SetBool("IsRunning", true);

break;

case CharacterLogic.allAnimations.Attack:

anim.SetTrigger("DoAttack");

anim.SetBool("IsRunning", false);

break;

case CharacterLogic.allAnimations.Dying:

anim.SetTrigger("IsDead");

break;

case CharacterLogic.allAnimations.GetHit:

anim.SetTrigger("GetHit");

anim.SetBool("IsRunning", false);

break;

}

}

}

**Листинг CharacterLogic.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CharacterLogic : MonoBehaviour, IEntity

{

public GameObject target = null;

public GameObject player = null;

public float speed = 2f;

public float rayDistance = 1f;

public float agressiveDistatnce = 5f;

private float healthPoints = 1000;

public float maxAngleVision;

public float maxRadiusVision = 10f;

private Rigidbody2D rb;

private float rightPatrolBorder;

private float leftPatrolBorder;

public static Vector2 movingDirection = Vector2.right;

public enum allAnimations { Staying, Running, Dying, Attack, GetHit };

private static allAnimations currentAnimation = allAnimations.Staying;

public static allAnimations CurrentAnimation { get => currentAnimation; set => currentAnimation = value; }

public float HealthPoints { get => healthPoints; set => healthPoints = value; }

void Start()

{

rb = GetComponent<Rigidbody2D>();

player = GameObject.FindGameObjectsWithTag("Player")[0];

rightPatrolBorder = transform.position.x + 8;

leftPatrolBorder = transform.position.x - 8;

Physics2D.queriesStartInColliders = false;

StartCoroutine(GoToMove());

}

IEnumerator GoToMove()

{

yield return new WaitForSeconds(Random.Range(0, 2));

currentAnimation = allAnimations.Running;

}

void Update()

{

if (currentAnimation != allAnimations.Dying)

{

Vector2 a = (player.transform.position - transform.position).normalized;

RaycastHit2D visionHit = Physics2D.Raycast(

transform.position,

(player.transform.position - transform.position).normalized,

maxRadiusVision);

if (visionHit.collider != null)

{

if (visionHit.collider.gameObject.tag == "Player")

{

if (player.transform.position.x > transform.position.x && movingDirection == Vector2.right)

{

target = player;

}

else if (player.transform.position.x < transform.position.x && movingDirection == Vector2.left)

{

target = player;

}

}

else

{

target = null;

}

}

else

{

target = null;

}

if (target == player)

{

if (player.transform.position.x > transform.position.x && movingDirection != Vector2.right)

{

TurnAround();

}

else if (player.transform.position.x < transform.position.x && movingDirection != Vector2.left)

{

TurnAround();

}

}

if (CurrentAnimation == allAnimations.Running)

{

transform.Translate(Vector2.right \* speed \* Time.deltaTime);

}

if (transform.position.x >= Random.Range(rightPatrolBorder - 2, rightPatrolBorder) && target == null && movingDirection == Vector2.right)

{

TurnAround();

}

else if (transform.position.x <= Random.Range(leftPatrolBorder, leftPatrolBorder + 2) && target == null && movingDirection == Vector2.left)

{

TurnAround();

}

RaycastHit2D hit = Physics2D.Raycast(transform.position, movingDirection, rayDistance);

if (hit.collider != null)

{

if (hit.collider.gameObject == player)

{

CurrentAnimation = allAnimations.Attack;

EndAttack();

}

else

{

TurnAround();

}

}

else

{

CurrentAnimation = allAnimations.Running;

}

}

}

public void Damage(float damage)

{

healthPoints -= damage;

Debug.Log(healthPoints);

if(healthPoints > 0)

{

CurrentAnimation = allAnimations.GetHit;

}

else

{

CurrentAnimation = allAnimations.Dying;

}

}

private void EndAttack()

{

Debug.Log("end attacs " + (player.transform.position.x - transform.position.x));

if (player.transform.position.x - transform.position.x < 1.7f )//&& player.transform.position.x - transform.position.x > 1.2f)

{

Debug.Log("aaa");

player.GetComponent<IEntity>().Damage(1);

}

}

private void EndGetHit()

{

CurrentAnimation = allAnimations.Staying;

}

private void EndDead()

{

CharacterAnim.anim.enabled = false;

}

private void TurnAround()

{

if (movingDirection == Vector2.right)

{

transform.eulerAngles = new Vector3(0, -180, 0);

movingDirection = Vector2.left;

}

else

{

transform.eulerAngles = new Vector3(0, 0, 0);

movingDirection = Vector2.right;

}

}

private void OnDrawGizmos()

{

Gizmos.color = Color.red;

Gizmos.DrawLine(transform.position, transform.position + Vector3.right \* rayDistance);

Gizmos.color = Color.red;

Gizmos.DrawLine(transform.position, transform.position + Vector3.left \* rayDistance);

Gizmos.color = Color.blue;

Gizmos.DrawLine(transform.position, transform.position + new Vector3(player.transform.position.x - transform.position.x, player.transform.position.y - transform.position.y, 0));

Gizmos.DrawWireSphere(transform.position, maxRadiusVision);

}

}

**Листинг HpController.cs:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class HPController : MonoBehaviour

{

private Slider slider;

private void Start()

{

slider = GetComponent<Slider>();

GameObject.FindGameObjectsWithTag("Player")[0].GetComponent<PlayerMovement>().UpdateHp += PlayerMovement\_UpdateHp;

}

private void PlayerMovement\_UpdateHp(float hpKf)

{

slider.value = hpKf;

}

}