ВВЕДЕНИЕ

Вместе с появлением компьютеров появились и компьютерные игры, которые сразу же нашли массу поклонников. Они являются неотъемлемой частью человеческой жизни с детства, таким образом, с одной стороны, снижают среднюю ежедневную двигательную активность и совместно проведенное со сверстниками время, которое, к тому же, как правило, сопровождается «живым» общением, а, напротив, способствуют развитию многих аспектов интеллектуальных способностей, в частности, логического и пространственного мышления, а также воображения человека. Когда человек учится жить в виртуальных мирах, быстро воспринимать незнакомые ситуации и адаптироваться к ним, он развивает свою интеллектуальную гибкость, тем самым обеспечивая свое приспособление к новым, неожиданным реалиям, которые могут возникнуть у него в жизни.

Таким образом, напрашивается вывод, что компьютерные игры, если не выполняют, то, как минимум, оказывают существенное влияние на функцию социализации молодежи в современном обществе.

Индустрия компьютерных игр появилась относительно недавно – порядка 30-ти лет назад, но уже успела превратиться в крупный сегмент индустрии развлечений, имеющий ежегодные доходы, исчисляемые несколькими десятками миллиардов долларов. Следствием такого бурного развития стало широкое распространение компьютерных технологий и появление сети Интернет. Компании создают игры под различные платформы, предоставляя возможность пользователю играть на персональном компьютере или игровой консоли дома и на мобильном устройстве в дороге. Ежегодно появляется все больше игр, которые поражают своей реалистичной графикой, живописным игровым миром, захватывающим сюжетом и уникальными механиками.

Из-за своей внушительной популярности, работа в сфере игровой индустрии является очень прибыльной уже на данный момент. По многочисленным прогнозам ведущих аналитиков, специалисты в данной области могут рассчитывать на свою востребованность в долгосрочной перспективе.

Целью полученного практического задания является разработка игрового приложения в жанре «*Arcada*» под платформу ПК.

1. ОБЩИЕ СВЕДЕНИЯ О ПРЕДПРИЯТИИ «*EPAM SYSTEMS*»

1.1 Описание организации

EPAM Systems – американская ИТ-компания, основанная в 1993 году. Производитель заказного программного обеспечения, специалист по консалтингу, резидент Белорусского парка высоких технологий. Штаб-квартира компании расположена в Ньютауне, штат Пенсильвания, а ее отделения представлены более чем в 40 странах мира [2].

Компания *EPAM* была основана в 1993 году двумя одноклассниками Аркадием Добкиным и Леонидом Лознером. Название компании происходило от «*Effective Programming for America*». Первые офисы были открыты в США и Белоруссии. Позже были открыты офисы в Австрии, Австралии, Армении, Болгарии, Великобритании, Венгрии, Германии, Индии, Ирландии, Казахстане, Канаде, Китае, Мексике, Нидерландах, ОАЭ, Польше, России, Сингапуре, Украине, Узбекистане, Чехии, Швеции, Швейцарии.

В 2021 году *EPAM Systems* заняла 1804 место в списке *Forbes Global* 2000 и была включена в *S&P* 500.

В 2013 году компания вошла в списки *Forbes* «25 самых быстрорастущих технологических компаний Америки» и «20 самых быстрорастущих технологических звезд Америки».

По данным «Коммерсанта», по итогам 2013 года *EPAM Systems* занимает десятое место в общем списке крупнейших ИТ-компаний России, первое место в списке разработчиков программного обеспечения, а также седьмое место в рейтинге крупнейших консультационных компаний России.

По данным рейтингового агентства «Эксперт РА» по итогам 2013 года компания возглавляет список ведущих разработчиков программного обеспечения, занимает восьмое место в рейтинге «Российский консалтинг» и десятое место в рейтинге крупнейших российских компаний в области ИКТ.

В 2014 году *EPAM* названа лидером в отчете «*The Forrester Wave*: *Software Product Development Services, Q*1 2014». По мнению *Forrester*, «*EPAM* опередила всех других поставщиков по продвижению инноваций и содействию в создании новых инновационных продуктов».

В 2015 году *Forrester* в своем исследовании выделил *EPAM* как одного из ключевых мировых поставщиков услуг в области *Agile*. Эксперты агентства особо отметили серьезные усилия компании по развитию *Agile*-тренингов, практик и инструментов.

Также 2015 году, в рейтинге «Коммерсанта» *EPAM* возглавляет список разработчиков ПО в России и входит в Топ-5 компаний российского ИТ-рынка, а также занимает шестое место среди крупнейших консультационных компаний в России.

По итогам 2016 года *EPAM* занял третье место в рейтинге *CNews* «Крупнейшие ИТ-компании России 2016», а также первое место среди

разработчиков ПО в рейтинге РБК+ российских ИТ-компаний (третье место в общем списке).

В 2016 году *EPAM* вошел в *TOP*-50 списка *CRN* «2016 *Solution Provider* 500».

В 2017 году *EPAM* снова включен в ежегодный рейтинг *Forbes* «25 самых быстрорастущих публичных технологических компаний Америки».

Также в 2017 *EPAM* вошел в список *IAOP* «*The Global Outsourcing* 100».

В 2018 году компания стала обладателем награды *MediaPost Appy Award* в категории «Путешествия и туризм», а также заняла 14 место в мировом рейтинге компаний, активно участвующих в *Open Source* проектах на *GitHub*.

В 2019 *EPAM* был удостоен награды *BIG Innovation Awards* за разработку продукта *TelescopeAI*, стал победителем международного конкурса *The Global SDG Awards* в номинации «Качественное образование», 12 сотрудников компании были признаны *Sitecore Most Valuable Professionals* (*MVPs*); компания также вошла в список 100 значимых компаний в сфере управления знаниями, опубликованный журналом *KMWorld Magazine*.

В 2021 *EPAM Systems* совместно с *Mars Incorporated* были объявлены победителями в номинации «*Leader of the Pack*: *Retail*» премии «*Acquia Engage Award* 2021». Наградами в данной номинации отмечаются цифровые решения и продукты, отличающиеся высоким уровнем функциональности, интеграции, производительности и удобства для пользователя.

1.2 Специализация организации, технологии и программное обеспечение, используемые на предприятии

Основная деятельность предприятия:

- ИТ-консалтинг;
- разработка программного обеспечения;
- интеграция приложений;
- портирование и миграция приложений;
- тестирование программного обеспечения;
- создание выделенных центров разработки на базе *EPAM Systems*;
- разработка цифровых стратегий.

Для контроля версий и передачи необходимых проектов и приложений используется система контроля версий *Git*. Системы контроля версий (СКВ, *VCS, Version Control Systems*) позволяют разработчикам сохранять все изменения, внесенные в код. При возникновении проблем они могут просто откатить код до рабочего состояния и не тратить часы на поиски ошибок. СКВ также позволяют нескольким разработчикам работать над одним проектом и сохранять внесенные изменения независимо друг от друга. При этом каждый участник команды видит, над чем работают коллеги.

В качестве клиент-сервер системы используется *GitHub*. *GitHub* – сервис онлайн-хостинга репозиториев, обладающий всеми функциями распределенного контроля версий и функциональностью управления исходным кодом – все, что поддерживает *Git* и даже больше [5]. Также *GitHub* может похвастаться контролем доступа, багтрекингом, управлением задачами и вики для каждого проекта. Таким образом обеспечивается общее взаимодействие всех сотрудников на предприятие с текущим проектом и распределение обязанностей.

Для коммуникации между членами организации используется специализированное программное обеспечение, имеющее название *Slack*. *Slack* является корпоративным мессенджером, запущенным в 2013 и ставшим самым быстрорастущим приложением. *Slack* собирает в одном окне обсуждения в общих темах (каналах), приватных группах и личных сообщениях; имеет собственный хостинг, режим предпросмотра изображений и позволяет искать среди всех сообщений сразу [6]. Кроме того, *Slack* поддерживает интеграцию с многочисленными сторонними сервисами, такими, например, как *Dropbox, Google Drive, GitHub, Google Docs, Google Hangouts, Twitter, Trello*.

На территории организации имеется выход в Интернет. Таким образом обеспечивается взаимодествие с программным обеспечением, требущим выхода в сеть, а именно: *Slack, Github*. Также Интернет на предприятии необходим для нахождения и использования специальной технической информации, требующейся для решения задач в рабочих проектах.

2. ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ РЕШЕНИЯ ЗАДАЧИ СОЗДАНИЯ ИГРОВОГО ПРИЛОЖЕНИЯ В ЖАНРЕ СТЕЛС «*HIDE*»

2.1 Предметная область решаемой задачи

Разработка игрового приложения будет осуществляться с помощью игрового движка *Unity*.

Unity – это кроссплатформенная среда разработки компьютерных игр, разработанная американской компанией *Unity Technologies*. *Unity* позволяет создавать приложения, которые работают на более чем 25-ти различных платформах, включая персональные компьютеры, игровые консоли, мобильные устройства, веб-приложения и многое другое [3]. *Unity* был выпущен в 2005 году и с тех пор постоянно развивается. Основными преимуществами *Unity* являются наличие визуальной среды разработки, кроссплатформенная поддержка и модульная компонентная система. К недостаткам можно отнести появление трудностей при работе с многокомпонентными схемами и трудности при подключении внешних библиотек. Тысячи игр, приложений, визуализаций математических моделей написаны на *Unity*, которые охватывают множество платформ и жанров.

Необходимо создать игровое приложение «» в жанре стелс.

Стелс (англ. *stealth* «невидимка; скрытность») – жанр компьютерных игр, в которых игрок должен избегать обнаружения игрового персонажа противниками или скрытно устранять их, не привлекая к себе внимания. Чтобы остаться незамеченным в этих играх, игровой персонаж должен прятаться от врагов, использовать маскировку и не создавать шума. Многие игры предоставляют игроку выбор – атаковать врагов открыто или сохранить скрытность; тем не менее, большинство игр поощряют игрока за сохранение максимально возможной степени скрытности [4]. В играх жанра используются мотивы шпионажа, контртерроризма и преступности. Протагонистами могут быть оперативники подразделений специального назначения, шпионы, воры, ниндзя, ассасины. В некоторых играх стелс-элементы комбинируется с геймплеем других жанров, например с шутерами от первого лица и даже с платформерами.

Уже в некоторых ранних играх появляется уклон в сторону скрытности, среди них *Manbiki Shounen* (1979), *Lupin III* (1980), 005 (1981), *Castle Wolfenstein* (1981), *Saboteur*! (1985), *Infiltrator* (1986), *Metal Gear* (1987), *Metal Gear* 2: *Solid Snake* (1990). Жанр обрел популярность в 1998 году, когда большого успеха добились игры *Metal Gear Solid*, *Tenchu*: *Stealth Assassins*, *Thief*: *The Dark Project*. *Tenchu* стала первой трехмерной игрой в жанре. Вышедший через несколько месяцев после нее *Metal Gear Solid* превратил малоизвестную до того серию *Metal Gear* в высоко оцениваемую и прибыльную франшизу с большим количеством последовавших сиквелов.

Игра *Thief*: *The Dark Project* стала первой игрой жанра стелс для *PC*. За этими играми последовали другие – *Hitman* и *Splinter Cell*. В выпущенных позднее играх предусмотрена возможность выбора между тактикой скрытности, открытой атакой или сочетанием их.

На рисунке 2.1 представлен скриншот из игры *Metal Gear Solid*.

Рисунок 2.1 – Скриншот из игры *Metal Gear Solid*

В отличие от большинства экшн-игр, в играх жанра стелс обычно необходимо избегать обнаружения врагом. Основными элементами геймплея современных стелс-игр являются разные способы уклониться от стычек с врагом, создание максимальной бесшумности, а также эффект неожиданности при атаке врагов из теней. Обычным делом в этих играх является выполнение заданий, оставаясь незамеченным врагом, что некоторые критики описывают как «призрачность». Несмотря на то, что в отдельных играх скрытность может быть единственным способом выиграть, в большинстве игр обычно предусматриваются другие способы и стили достижения поставленной задачи. Игроки могут прятаться за предметами или скрываться в тенях, а при встрече с врагом могут или атаковать его, или незаметно проскользнуть мимо. Если игрок будет обнаружен врагом, обычно требуется спрятаться на некоторое время, пока враги не прекратят поиск. Таким образом, важность приобретают планирование и метод проб и ошибок. Однако, в некоторых стелс-играх придается важность мастерству рукопашного боя, которое становится полезным в случае обнаружения персонажа врагом. В некоторых играх игрок может выбирать между убийством врагов или простым его оглушением. Если «призрачность» необязательна для прохождения, либо в недостаточной степени реализована в игре, игроки все же могут пробовать избегать сражений по моральным причинам или для демонстрации своего мастерства.

Поскольку возможность скрываться в тенях – элемент геймплея, грамотная работа со светом и тенью является требованием к дизайну уровней. Обычно у игрока есть возможность отключать некоторые источники света. Также для стелс-игр важным является звуковое

сопровождение, чтобы игрок мог услышать еле заметные звуки, которые могут заметить враги и предпринять некоторые ответные действия. Как правило в этих играх производимый игроком шум будет отличаться в зависимости от поверхности, по которой он идет, например, по металлу или дереву. Беспечные перемещения игрока создают больше шума, чем привлекают внимание врагов.

Для включения в геймплей игры стелс-элементов необходимо ограничить осведомленность искусственного интеллекта (ИИ) таким образом, чтобы он не знал об определенных частях игры. В стелс-играх ИИ принимает определенные решения относительно действий врагов в ответ на результаты действий игрока, такие как отключение освещения, вместо прямой реакции на действия игрока. У врагов обычно есть некий угол обзора, поэтому игрок, чтобы не быть замеченным, может прятаться за предметами, скрываться в тенях или перемещаться, пока враг смотрит в другую сторону. Как правило, враги могут обнаруживать, когда игрок дотрагивается до них или находится на определенном незначительном расстоянии. В целом, то, на какие действия игрока будет реагировать ИИ, в разных играх различно, причем в более современных играх можно найти более широкий спектр вражеских ответных действий. Часто движения управляемых ИИ врагов шаблонны и предсказуемы, что позволяет придумать стратегию для их преодоления. В играх жанра стелс игрок обычно ограничен в возможностях участия в непосредственном сражении с врагом, поскольку обычно персонаж имеет при себе неэффективное оружие или оружие несмертельного действия, в то время как враги превосходят игрока как в вооруженности, так и в количестве. Также у персонажа может быть ограничено количество очков здоровья, что делает любые непосредственные боевые столкновения крайне опасными. Иногда жанр стелс смешивается с жанром «выживание в кошмаре», в котором игрок должен прятаться и избегать встреч со сверхъестественными или, редко, обычными земными врагами, несмотря на их попытки выследить игрока. Примерами такого смешения жанров *Stealth*/*Horror* (с англ. – «скрытность и кошмар») являются серии игр *Amnesia*: *The Dark Descent, Outlast* и *Penumbra*.

Следовательно, разрабатываемое игровое приложение «*Hide*» должно включать в себя следующее:

– главное меню;

– нелинейный уровень (уровни) для исследования (прохождения);

– несколько разновидностей врагов, которые отличаются между собой поведением относительно игрока и друг друга;

– игрок должен иметь возможность превращаться в каждый из типов врагов при необходимости;

– другие различные игровые механики, характерные для жанра *Stealth*.

Разработка игрового приложения будет осуществляться с использованием следующих технологий: среды разработки *Microsoft Visual*

Studio 2022 *Community Edition* и ее компонентов, языка программирования *C#*, игрового движка *Unity* крайней актуальной версии.

2.2 Применение метода декомпозиции при проектировании игрового приложения «*Hide*»

При проектировании игрового приложения «*Hide*» было принято решение использовать метод декомпозиции.

Декомпозиция – операция мышления, состоящая в разделении целого на части. Также декомпозицией называется общий прием, применяемый при решении проблем, состоящий в разделении проблемы на множество частных проблем, а также задач, не превосходящих суммарно по сложности исходную проблему, с помощью объединения решений которых, можно сформировать решение исходной проблемы в целом [7].

Впервые в литературе в явной и отчетливой форме декомпозиция (деление трудностей (*difficulties*) на части), была рассмотрена Р. Декартом в перечне из четырех базовых правил решения проблем («трудностей»), в работе «Рассуждение о методе», ознаменовавшим переход к современному научному познанию.

Декомпозиция, как процесс расчленения, позволяет рассматривать любую исследуемую систему как сложную, состоящую из отдельных взаимосвязанных подсистем, которые, в свою очередь, также могут быть расчленены на части. В качестве систем могут выступать не только материальные объекты, но и процессы, явления и понятия.

В общем виде, как операция мышления декомпозиция является обратной к операциям абстрагировании и обобщения.

При декомпозиции руководствуются следующими правилами:

– каждое расчленение образует свой уровень;

– система расчленяется только по одному, постоянному для всех уровней, признаку. В качестве признака декомпозиции может быть: функциональное назначение частей, конструктивное устройство (например, вид материалов, формы поверхностей), структурные признаки (например, вид схемы, способы), виды этапов и процессов (например, жизненный цикл, физическое состояние и), предметные характеристики (например, экономические, информационные, технологические);

– вычленяемые подсистемы в сумме должны полностью характеризовать систему. Но при этом вычленяемые подсистемы должны взаимно исключать друг друга. Например, если при перечислении частей автомобиля опустить, допустим, мотор, то функциональное взаимодействие остальных подсистем не обеспечит нормальное функционирование всей системы (автомобиля) в целом. В другом примере, перечисляя возможные виды двигателей, используемые в автомобиле, необходимо охватить всю известную область (декомпозиция – по принципу действия). Если это сложно сделать, допускается неупомянутые (или неизвестные) элементы объединить в одну группу (подсистему) и назвать ее «другие», либо «прочие», либо

провести деление двигателей, например, на «тепловые» и «нетепловые». К неоднозначности может привести использование на одном уровне взаимно пересекающихся подсистем, например, «двигатели электрические» и «двигатели переменного тока», так как неясно куда же нужно в таком случае отнести асинхронный двигатель. Для обозримости рекомендуют выделять на каждом уровне не более семи подсистем. Недопустимо, чтобы одной из подсистем являлась сама система.

Степень подробности описания и количество уровней определяются требованиями обозримости и удобства восприятия получаемой иерархической структуры, ее соответствия уровням знания работающему с ней специалисту.

Обычно в качестве нижнего (элементарного) уровня подсистем берут такой, на котором располагаются подсистемы, понимание устройства которых или их описание доступно исполнителю (руководителю группы людей или отдельному человеку). Таким образом, иерархическая структура всегда субъективно ориентирована: для более квалифицированного специалиста она будет менее подробна.

Число уровней иерархии влияет на обозримость структуры: много уровней – задача труднообозримая, мало уровней – возрастает число находящихся на одном уровне подсистем и сложно установить между ними связи. Обычно, в зависимости от сложности системы и требуемой глубины проработки, выделяют от трех до шести уровней.

Например, разрабатывая механический привод, в качестве элементарного уровня можно взять колеса, валы, подшипники, двигатель в целом. Хотя подшипники и двигатель являются сложными по устройству элементами и трудоемкими в проектировании, но как готовые покупные изделия для разработчика они выступают в виде элементарных частей. Если бы двигатель пришлось бы разрабатывать, то его, как сложную систему, было бы целесообразно декомпозировать.

2.3 Структура игрового приложения «*Hide*»

Таким образом, метод декомпозиции позволяет разделить искомое игровое приложение на несколько более компактных, связанных между собой проектов, а именно: игровой движок, скрипты, описывающие игровую логику, пользовательский интерфейс, скрипты для работы со звуком.

Игровой движок (англ. *game engine*) – базовое программное обеспечение компьютерной игры. Разделение игры и игрового движка часто расплывчато, и не всегда студии проводят четкую границу между ними. Но в общем случае термин «игровой движок» применяется для того программного обеспечения, которое пригодно для повторного использования и расширения, и тем самым может быть рассмотрено как основание для разработки множества различных игр без существенных изменений.

Термин «игровой движок» появился в середине 1990-х в контексте компьютерных игр жанра шутер от первого лица, похожих на популярную в

то время *Doom*. Архитектура программного обеспечения *Doom* была построена таким образом, что представляла собой разумное и хорошо выполненное разделение центральных компонентов игры (например, подсистемы трехмерной графики, расчета столкновений объектов, звуковой и других) и графических ресурсов, игровых миров, формирующие опыт игрока, игровые правила и другое. Как следствие, это получило определенную ценность за счет того, что начали создаваться игры с минимальными изменениями, когда при наличии игрового движка компании создавали новую графику, оружие, персонажей, правила игры и тому подобное.

Разделение между игрой и игровым движком часто неопределенно. Некоторые движки имеют разумное и ясное разделение, в то же время другие практически невозможно отделить от игры. Например, в игре движок может «знать» о том, как рисовать дугу, в то же время другой движок может работать с другим уровнем абстракции, и в нем дуга будет частным случаем параметров вызываемых функций [8]. Одним из признаков игрового движка является применение архитектуры управления данными. Это определяется тем, что если игра содержит жестко фиксированные данные, влияющие на логику, правила игры, рисование объектов и тому подобное, то становится сложно применять данное программное обеспечение в разных играх.

Большинство игровых движков разработано и настроено для того, чтобы запустить определенную игру на определенной платформе. И даже наиболее обобщенные многоплатформенные движки подходят для построения игр определенного жанра, например, шутеров первого лица или гонок. В данном контексте можно более аккуратно сказать, что игровой движок становится не оптимальным при его применении не для той игры или той платформы, для которой разработан. Данный эффект проявляется от того, что программное обеспечение представляет собой набор компромиссов, основанных на тех предположениях, какой должна быть игра. Например, проектирование рендеринга внутри зданий приведет к тому, что движок, скорее всего, не будет таким же хорошим для открытых пространств. В первом случае движок может использовать *BSP*-дерево для отрисовки объектов, близких к камере. В то же время для открытых пространств могут использоваться менее точные способы, а также более активно применяются технологии отрисовки с разной степенью детализации, когда более далекие объекты прорисовываются менее четко, так как занимают меньшее количество пикселей.

Как правило, игровые движки специализированы в рамках жанра компьютерных игр. Так, движок, спроектированный для двумерного файтинга на боксерском ринге, будет существенно отличаться от движка для массовой многопользовательской игры, шутера от первого лица или стратегии в реальном времени [9]. Но в то же время движки имеют существенные общие части – все трехмерные игры, невзирая на жанр, требуют взаимодействия игрока посредством клавиатуры, геймпада и/или мыши, некоторую форму трехмерного рендеринга, средства индикации, как на лобовом стекле (например, печать текста поверх графического

изображения), звуковую систему и многое другое. Так, движок *Unreal Engine*, несмотря на то, что был спроектирован для шутера от первого лица, успешно использовался для создания игр во множестве других жанров, таких как шутер от третьего лица *Gears of War*, приключенческая ролевая игра *Grimm*, или футуристичная гонка *Speed Star*.

Исторически шутеры от первого лица относятся к играм, которые наиболее технологически сложны, так как им необходимо представлять игроку иллюзию трехмерного мира, и делать это для активных действий в реальном времени. Движки шутеров от первого лица больше обращают внимание на такие технологии, как эффективный рендеринг трехмерных миров, отзывчивая игровая механика контроля и прицеливания, высокая точность анимации оружия и рук управляемого игроком персонажа, широкий спектр ручного вооружения, «прощающая» модель движения игрока и его столкновения с препятствиями, высокое качество анимации и искусственного интеллекта неигровых персонажей [10]. При этом характерны малая масштабируемость в многопользовательских играх (типична поддержка до 64 игроков) и повсеместная ориентация на игровой процесс *deathmatch*. Графические движки игр данного жанра используют ряд оптимизаций в зависимости от текущего окружения игрока, но вместе с тем предъявляются требования по анимации персонажа, аудио и музыке, динамики твердого тела, кинематике и другим технологиям.

Движки платформеров обращают больше внимания на анимацию персонажа и его аватара, и при этом им не требуется той реалистичности, которая присуща трехмерным шутерам. Для платформеров характерно применение ряда технологий: множество способов перемещения (движущиеся платформы, лестницы, веревки, подпорки и другие), элементы из головоломок, использование следящей за персонажем камеры от третьего лица, рендеринг нескольких слоев геометрии в сочетании с системой столкновений объектов, и другие.

Файтинги ориентированы на богатую анимацию, точность ударов, возможность задания сложных комбинаций посредством кнопок и/или джойстика и тому подобное. Анимационные персонажи предъявляют требования движкам по высокой детализации, дополнительно движки обеспечивают возможность изменения и добавления спецэффектов (шрамов после ударов, выступление пота и тому подобное), а также предоставляются возможности симуляции прически, одежды и других элементов.

Автосимуляторы могут быть разными, и здесь имеется ряд поджанров. Графика таких игр ориентирована на «коридорность» и кольцевые треки, и поэтому движки больше обращают внимание на детализацию машин, трека и непосредственное окружение. Как следствие, используются технологии для рендеринга далеких фоновых объектов (отображаемых двумерно), трек часто разделяется на несколько секторов, внутри которых проводится оптимизация по рендерингу. В случае движения по туннелям или другим «тесным» местам используются техники для того, чтобы камера с видом от третьего лица не пересекалась с фоновой геометрией. Используемые структуры данных и

искусственный интеллект ориентируются на решение задач машин неигровых персонажей, таких как поиск пути и других технических проблем.

У стратегий реального времени нет высоких требований к графике, и поэтому движок ориентируется на то, что отображает юнитов в низком разрешении, но при этом он должен быть способен работать с большим числом юнитов одновременно. Отдельные особенности имеются у интерфейса взаимодействия игрока и элементов управления, в которые входят инструменты работы с группами юнитов (выделение по площади, управление) и ряд меню и панелей инструментов, содержащих команды управления, элементы снаряжения, выбор типов юнитов и зданий и тому подобное.

Массовые многопользовательские игры требуют наличия большого игрового мира и возможности одновременного присутствия и взаимодействия большого числа игроков. Локальные задачи, решаемые движком, похожи на те, что имеются в играх других жанров, но особенностью жанра является ориентация и проработка программного обеспечения серверов, которые должны сохранять состояние мира, управлять подключением и отключением игроков, предоставлять внутриигровые чаты, способы взаимодействия голосом и так далее.

Проект игровой логики, используя возможности игрового движка путем реализации игровых компонентов, а также задействования игровых ресурсов и менеджера для них, представляет собой некоторый менеджер, в нашем случае – игровой. Он отвечает за порядок выполнения и непосредственно выполение таких действий, как создание сцены, персонажей; добавление и удаление элементов; отрисовку всего вышеперечисленного. Игровая логика содержит готовые необходимые игровые объекты и скрипты, описывающие поведение игровых объектов, а также экземпляры сцен.

Интерфейс пользователя, он же пользовательский интерфейс (*UI* – англ. *user interface*) – интерфейс, обеспечивающий передачу информации между пользователем-человеком и программно-аппаратными компонентами компьютерной системы.

Под совокупностью средств и методов интерфейса пользователя подразумеваются средства и методы.

Средства:

– вывода информации из устройства к пользователю – весь доступный диапазон воздействий на организм человека (зрительных, слуховых, тактильных, обонятельных и т. д.) – экраны (дисплеи, проекторы) и лампочки, динамики, зуммеры и сирены, вибромоторы и т. д. и т. п.

– ввода информации/команд пользователем в устройство – множество всевозможных устройств для контроля состояния человека – кнопки, переключатели, потенциометры, датчики положения и движения, сервоприводы, жесты лицом и руками, даже съем мозговой активности пользователя.

По наличию тех или иных средств ввода, интерфейсы разделяются на типы – жестовый, голосовой, брэйн, и т. д., возможны смешанные варианты. Средства эти должны быть необходимыми и достаточными, быть удобными и практичными, расположенными и скомпонованными разумно и понятно, соответствовать физиологии человека, не должны приводить к негативным последствиям для организма пользователя (все это входит в понятие эргономики).

Методы: набор правил, заложенных разработчиком устройства, согласно которым совокупность действий пользователя должна привести к необходимой реакции устройства и выполнения требуемой задачи – так называемый логический интерфейс.

В случае с разрабатываемым игровым приложением, проект пользовательского интерфейса следует отделить от остального программного кода не только для удобства и наилучшей читабельности, а также по той причине, что проект также будет довольно насыщенным. Необходимо реализовать не только главное меню, но и некоторые всплывающие окна для взаимодействия пользователя с приложением в процессе игры, вывод текстовой информации (*TextMeshProUGUI*), кнопки, информационные панели (состояние игрока). Также подразумевается возможность кастомизации вышеперечисленного функционала данного проекта (стилизованный текст, цветной *HUD*, фоновые изображения)

В свою очередь, скрипты для работы со звуком будут вынесены в отдельный проект преимущественно для удобства и лучшей читабельности кода. Необходимо реализовать звуки при нажатии на кнопки в главном меню, а также во время игрового процесса

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «*HIDE*»

3.1 Апробация игрового приложения «*Hide*»

Целью игры прохождение уровня

После запуска исполняемого файла программы пользователь попадает в ее главное меню, которое предлагает следующие опции:

– «*Continue*» – продолжить уже начатое прохождение (если таковое имеется).

– «*New Game*» – начать новую игру.

– «*Options*» – игровые настройки.

– «*Quit*» – выход из игры и закрытие приложения с сохранением прогресса.

На рисунке 3.1 представлен скриншот главного меню игрового приложения «*Hide*».

Рисунок 3.1 – Скриншот главного меню игрового приложения «*Hide*»

В случае нажатия на кнопку «*Quit*», процесс выполнения программы завершается, для повторного запуска необходимо снова открыть исполняемый файл программы.

В случае нажатия на кнопку «*Play*», пользователь попадает на сцену с игровым уровнем. После начала игры уровень уже полностью сгенерирован. Игрок появляется в определенной точке в начале уровня. Ему необходимо добраться к концу уровня, раположение которого неизвестно – его необходимо выяснить опытным путем, преодолевая препятствия, решая головоломки и исбегая стычек с врагами. Персонажем по умолчанию для игрока является человек. Врагами для игрока на уровне являются зомби и вооруженные охранники. Игрок обладает такими базовыми игровыми механиками, как ходьба, бег, а также может красться. При необходимости, игрок может превращаться в зомби, таким образом, другие зомби не будут его атаковать, если он с ними столкнется, но в таком случае ему будет сложнее пройти охранников, которые агрессивно реагируют на зомби в не зависимости от их поведения, расстояния, на котором они находятся и т.д., на человека же охранники реагируют в зависимости от скорости его передвижения, а также не обращают на него внимания на большом расстоянии. Поэтому игроку необходимо грамотно подходить к тому, когда ему стоит превратиться в зомби – и обратно. Также, что касается зомби: они имеют сравнительно низкую скорость перемещения, способны атаковать только вплотную и не обращают внимания на игрока, если тот крадется или превратился в зомби, пускай даже прямо у них на виду. Охранники, в свою очередь, обладают более сложным поведением: они могут атаковать как зомби, так и игрока при помощи стрельбы, точность которой является

непредсказуемой, а также способны обратить внимание на игрока вне зависимости от того, каким образом он перемещается – крадется или идет/бежит в полный рост. Что касается механик перемещения для персонажей: зомби имеют полную свободу перемещения в рамках уровня, движутся в случайном направлении (в направлении случайно сгенерированной «на лету» точки, от точки к точке и т. д.) с постоянной скоростью, каждый из охранников движется по своей заданной траектории (по предварительно сгенерированным точкам, туда и обратно), охранники не обладают такой же «свободой перемещения», как у зомби, то есть они всегда находятся в определнных локациях на уровне и не могут их покинуть, игрок же обладает «полной свободой перемещения» в рамках уровня.

На рисунке 3.2 представлены скриншоты игрового процесса.

Рисунок 3.2 – Скриншоты игрового процесса

В игре реализовано следующее звуковое сопровождение:

– звуки при нажатии на кнопки интерфейса;
– звуки перемещения для каждого из видов персонажей;
– стандартные звуки зомби (*idle*, при атаке игрока, получении урона и т. д.);
– стрельба;
– фоновая музыка в главном меню и на игровом уровне.

В игре также реализовано сохранение прогресса пользователя.

3.2 Структура данных игрового приложения «*Hide*»

В проекте игрового приложения «*Hide*» реализованы следующие скрипты:

– *animationPlayer* – логика работы анимация для игрока;
– *attackBehaviourSwat* – логика работы и параметры атаки для охранника;
– *attackBehaviourZombie* – логика работы и параметры атаки для зомби;
– *attackRunBehaviourSwat* – логика работы и параметры атаки для охранника при беге ;
– *chaseBehaviourSwat* – логика работы и параметры преследования цели охранником;
– *chaseBehaviourZombie* – логика работы и параметры преследования цели зомби;
– *checkDetectPlayer* – скрипт обнаружения игрока;
– *detectEnemy* – логика работы обнаружения игрока врагами;
– *gameData* – текущее состояние прохождения уровня;
– *gameManager* – игровой менеджер;

– *gameProgress* – скрипт для сохранения и загрузки игрового прогресса;

– *health* – логика работы здоровья игровых персонажей;

– *iCheckDetect* – интерфейс для *checkDetect*;

– *idleBehaviourSwat* – поведение охранника в состоянии простоя;

– *idleBehaviourZombie* – поведение зомби в состоянии простоя;

– *iHealth* – интерфейс для *health*;

– *movePlayer* – скрипт для перемещения игрока;

– *moveState* – текущее состояние игрока при перемещении;

– *moveWithMouse* – скрипт для перемещения игрока при помощи мыши;

– *iMove* – интерефейс для *moveState*;

– *PersuitEnemy* – скрипт, предназначенный для того, чтобы враги различали игрока и друг друга, и предпринимали соответствующие действия;

– *iPersuit* – интерфейс для *PersuitEnemy*;

– *level* – скрипт, описывающий игровой уровень;

– *loader* – скрипт для начала игры (инициализирует объект игрового менеджера, осуществляет появление игрока на уровне);

– *mainMenu* – скрипт для работы кнопок главного меню;

– *nextSceneManager* – осуществляет загрузку следующей сцены;

– *patrolBehaviour* – поведение охранника при патрулировании территории;

– *pauseMenu* – логика работы кнопок меню паузы;

– *player* – скрипт, описывающий игрока;

– *patrolBehaviourZombie* – поведение зомби при исследовании местности;

– *pulseColor* – блики на экране при получении урона или превращении;

– *reincarnation* – скрипт для осуществления респавна;

– *saveAndLoad* – скрипт для записи в файл игрового прогресса, а также загрузки и распаковки данного файла при продолжении игры из главного меню;

– *swat* – скрипт, описывающий охранника;

– *uiHelper* – логика работы *UI*;

– *weapon* – логика работы огнестрельного оружия охранника;

– *zombiAI* – *AI* (поведение) зомби.

ЗАКЛЮЧЕНИЕ

Результатом проделанной работы является разработанное под платформу ПК игровое приложение «*Hide*», которое включает в себя следующее:

– главное меню;

– нелинейный уровень (уровни) для исследования (прохождения);

– несколько разновидностей врагов, которые отличаются между собой поведением относительно игрока и друг друга;

– игрок должен иметь возможность превращаться в каждый из типов врагов при необходимости;

– другие различные игровые механики, характерные для жанра *Stealth*.

Для программной реализации приложения использовались: средства языка программирования *C#*, игровой движок *Unity*.

При разработке игрового приложения был проведен аналитический обзор игр, относящихся к жанрам «*Maze*», «*Shooter*» и «*Stealth*», что может подразумевать наличие средств разработки, достаточных для освоения заданной темы.

Определены структура и алгоритм разработки программного обеспечения. Ключевые этапы разработки игрового приложения «*Hide*» – с использованием возможностей игрового движка *Unity* создание игровой логики, пользовательского интерфейса, скриптов для работы со звуком. Разработанные скрипты протестированы, ошибки, выявленные по результатам тестирования, устранены, что гарантирует корректную работу приложения и минимизирует вероятность получения пользователем ошибки.

Игра имеет минимальный порог вхождения, интуитивно понятна и, как следствие, не требует специальных навыков. Динамичный геймплей гарантирует полное погружение в игровой процесс, получение интересного опыта, положительно сказываясь на таких спектрах развития, как логическое мышление и внимательность, что в совокупности повышает значимость и ценность игры как программного продукта.

Приложение прошло опытную эксплуатацию с положительным результатом – ошибки и недочеты учтены и устранены. Техническая документация к проекту и сам проект проверены системой «Антиплагиат», оригинальность составила 90.08%.

Авторские права на программную часть проекта принадлежат автору курсовой работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *The videogame style guide and reference manual* / Д. Томас [и др.]. – *Lulu.com*, 2007. – 108с.

2. *EPAM Systems* [Электронный ресурс]. – 2022. – Режим доступа: *https://ru.wikipedia.org/wiki/EPAM_Systems*/. – Дата доступа: 02.09.2022.

3. *Unity* [Электронный ресурс]. – 2022. – Режим доступа: *https*://*ru.wikipedia.org*/*wiki*/*Unity*_(%*D*0%*B*8%*D*0%*B*3%*D*1%80%*D*0%*BE*%*D*0%*B*2%*D*0%*BE*%*D*0%*B*9_%*D*0%*B*4%*D*0%*B*2%*D*0%*B*8%*D*0%*B*6%*D*0%*BE*%*D*0%*BA*)/. – Дата доступа: 02.09.2022.

4. Стелс (компьютерные игры) [Электронный ресурс]. – 2022. – Режим доступа: *https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BB%D1%81_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D0%B5_%D0%B8%D0%B3%D1%80%D1%8B)*/. – Дата доступа: 02.09.2022.

5. *Github* [Электронный ресурс]. – 2022. – Режим доступа: *https*://*ru.wikipedia.org*/*wiki*/*GitHub*/. – Дата доступа: 02.09.2022.

6. *Slack* [Электронный ресурс]. – 2022. – Режим доступа: *https*://*ru.wikipedia.org*/*wiki*/*Slack*/. – Дата доступа: 02.09.2022.

7. Цурков В.И. «Декомпозиция в задачах большой размерности». Под ред. Г.С. Поспелова. 1981. – 352 с.

8. *Jason, Gregory*. «*Game Engine Architecture*» / *J. Gregory*. – «*CRC Press*», 2009. – 864 *p.*

9. *Rouse, R.* «*Game Design*: *Theory and Practice*» / *R. Rouse*. – *Texas*: «*Los Rios Boulevard Plano*», 2016. – 723 *p.*

10. «Компьютерные миры *ZX Spectrum*». – Санкт-Петербург: «Питер», 1995. – Т. 2. – С. 7–8. – 221 с.

ПРИЛОЖЕНИЕ А

(обязательное)

Охрана труда и техника безопасности на рабочем месте программиста

Сегодня профессия программист особенно востребована. Ни одно производство или сфера услуг не обходится без компьютера Современный офис тяжело представить без компьютера. Тем более если речь идет об *IT*-компании, осуществляющей деятельность в области программного обеспечения. На первый взгляд работа кажется безопасной, Но если разобраться, то становится понятно, что этот специалист регулярно подвергается воздействию излучения, электричества, страдает от запыленности воздуха рабочей зоны, шума и вибрации, яркости изображения монитора, что негативно влияет как на зрение, так и на психо-эмоциональное состояние. Потому программист обязан следовать технике безопасности на рабочем месте на всех этапах своей деятельности.

Основными нормативными правовыми актами по охране труда при работе с компьютерами являются:

Типовая инструкция по охране труда при работе с персональными электронными вычислительными машинами (постановление Министерства труда и социальной защиты Республики Беларусь от 24.12.2013 № 130).

Санитарные нормы и правила «Требования при работе с видеодисплейными терминалами и электронно-вычислительными машинами», Гигиенический норматив «Предельно допустимые уровни нормируемых параметров при работе с видеодисплейными терминалами и электронно-вычислительными машинами» (постановление Министерства здравоохранения Республики Беларусь от 28.06.2013 № 59) (далее – СанПиНиГН № 59).

Основные аспекты охраны труда программиста перед началом работы. Перед тем, как приступить к работе за компьютером, программист обязан:

- Осмотреть свое рабочее место.
- Очистить его от посторонних предметов и захламленности.
- Проверить освещение рабочей зоны и довести ее до нужных показателей.
- Проверить заземление компьютера.
- Проверить включение компьютера к сети.
- Протереть поверхность монитора влажной тканью.
- Проверить уровень стула, стола.
- Программист не должен начинать работу при несоответствии параметров техники всем нужным требованиям санитарных норм.
- При отсутствии фильтра класса «полная защита».
- При обнаружении поломки оборудования.
- При отсутствии вблизи рабочей зоны первичных средств пожаротушения и аптечки.

Основные аспекты охраны труда программиста во время работы. Во время своей рабочей деятельности программист должен:

– Выполнять лишь тот отрезок работы, который ему поручен руководством.

– Следить за чистотой рабочего места и проходами возле него.

– Все вентиляционные отверстия в технике должны быть открыты.

– Если есть необходимость отойти от рабочего места, программист должен закрыть все ненужные активные окна.

– Соблюдать режим труда и отдыха.

– Соблюдать выделенные перерывы в работе, делать регулярные упражнения для глаз, шеи, рук, ног.

– Не переутомляться за компьютером.

– Программист не должен трогать технику мокрыми или влажными руками.

– Не должен самостоятельно заниматься ремонтом или разборкой техники.

– Не должен захламлять поверхность монитора или системный блок посторонними предметами.

– Не должен пить или есть за столом, где находится техника.

– Не должен отключать питание во время активного или малоактивного использования компьютера.

Основные аспекты охраны труда программиста после окончания трудовой деятельности. Закончив работу за компьютером, программист обязан:

– Закрыть все активные окна на компьютере.

– Отключить процессор от сети.

– Отключить от питания все дополнительные приборы.

– Отключить блок питания.

– Убрать рабочее место от мусора и бумаг.

– Вымыть руки с мылом.

При повреждении оборудования, кабелей, проводов, неисправности заземления (зануления), появлении запаха гари, возникновении необычного шума и других неисправностях программист обязан немедленно отключить электропитание оборудования и сообщить о случившемся непосредственному руководителю или иному уполномоченному должностному лицу нанимателя.

При несчастном случае необходимо немедленно сообщить о несчастном случае непосредственному руководителю или иному уполномоченному должностному лицу и принять меры по предотвращению воздействия травмирующих факторов. В случае получения травмы и (или) внезапного ухудшения здоровья (усиления сердцебиения, появления головной боли и другого) нужно прекратить работу, выключить оборудование, сообщить об этом непосредственному руководителю или иному уполномоченному должностному лицу и при необходимости обратиться к врачу.

Такие простые правила помогут программисту легко и эффективно провести свой рабочий день, не нанести ущерб своему здоровью и не

переутомить глаза, а также избежать травм и поражений током на работе. Их соблюдение должно быть правилом, действиями, доведенными до автоматизма, чтобы свести риски до минимума.

Техника безопасности при работе с компьютером. Вредных факторов при работе за компьютером есть достаточно много. Все они в разной степени влияют на организм, создавая угрозу его правильному функционированию. Среди таких факторов выделяют:

- Нагревание деталей и поверхности компьютера в процессе активной эксплуатации, что приводит к повышению уровня температуры в помещении.
- Высокая зрительная нагрузка при продолжительной работе.
- Монотонность рабочего процесса.
- Риск поражения электрическим током при неправильной эксплуатации или игнорировании фактов поломки техники.
- Ухудшение зрения при условии плохого освещения рабочей зоны.
- Высокий уровень контраста и блеска экрана, что пагубно влияет на зрение работника.

Требования к работе за компьютером. Основные требования, которые прописаны в документации по безопасности труда при работе за компьютером, затрагивают многие вопросы, которые напрямую влияют на состояние нашего организма. Основные требования гласят:

- Следование общим правилам использования компьютерной техники.
- Соблюдение микроклимату помещения, в котором проводиться работа за компьютером.
- Соблюдение уровня шума, который исходит от компьютерной техники.
- Соблюдение уровня освещения при работе с компьютерной техникой.
- Соблюдение нормы электромагнитных полей.
- Прохождение медицинской комиссии работниками, деятельность которых связана с регулярной работой за компьютером.

Меры безопасности перед началом работы за компьютером. До начала работы нужно:

- Проверить исправность розеток, проводов и кабелей компьютера.
- Убедиться в заземлении техники.
- Проверить, работает ли компьютер.

Меры безопасности во время работы за компьютером.

- Не трогать компьютер и системный блок мокрыми руками.
- Не класть на корпус компьютера посторонние предметы.
- Самостоятельно чистить системный блок от пыли и грязи, при условии его подключения к напряжению.
- Если обнаружили неполадки в работе компьютера – немедленно прекратить работу за ним и сообщите об этом руководителю.
- При эксплуатации компьютера, соблюдать все рекомендации производителя.

– Не включать или выключать компьютер из розетки без особой надобности и часто во время рабочего процесса.

Меры безопасности после окончания работы за компьютером. После завершения работы за компьютером, работник обязан:

– Выключить компьютер из сети, соблюдая все правила безопасности.
– Убедиться в том, что компьютер не работает.
– При помощи влажной ткани очистить поверхность техники.

Все вышеперечисленные меры безопасности помогут максимально эффективно использовать компьютер в рабочей деятельности и не причинить вреда своему здоровью и здоровью коллег.

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программы

Листинг программы для скрипта *AnimationPlayer.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AnimationPlayer : MonoBehaviour
{
    private Animator animator;

    public IMove moveState;

    // Start is called before the first frame update
    void Start()
    {
        animator = Player.instance.animator;
        moveState = Player.instance.moveState;
    }

    public void SetAvatar(Avatar avatar)
    {
        this.animator.avatar = avatar;
    }

    // Update is called once per frame
    void Update()
    {
        animator.SetFloat("Speed", (moveState.Magnitude * moveState.CurrSpeed) / moveState.Speed);

        if (moveState.Magnitude != 0 && !moveState.IsSneakWalk)
        {
            animator.SetBool("IsRunning", moveState.IsRunning);
        }
        else
        {
            animator.SetBool("IsRunning", false);
        }

        if (moveState.Magnitude != 0)
        {
            animator.SetBool("IsSneakWalk", moveState.IsSneakWalk);
        }
        else
        {
            animator.SetBool("IsSneakWalk", false);
        }

        animator.SetBool("IsDeath", !moveState.IsCanMove);
    }
}
```

Листинг программы для скрипта *AttackBehaviourSwat.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AttackBehaviorSwat : StateMachineBehaviour
{
    float attackMinRange = 5;
    float attackMaxRange = 10;
    float chaseRange = 15;

    float timer = 0;

    private PursuitEnemy pursuitEnemy;

    private Weapon weapon;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
```

```
    {
        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
        weapon = animator.GetComponent<Swat>().weapon; ;
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer += Time.deltaTime;

        if (!pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsShoot", false);
            animator.SetBool("IsChase", false);
            return;
        }

        animator.transform.LookAt(pursuitEnemy.enemy.transform);
        float distance = Vector3.Distance(animator.transform.position, pursuitEnemy.enemy.transform.position);

        if (distance > attackMinRange && distance < chaseRange)
            animator.SetBool("IsChase", true);

        if (distance > attackMaxRange)
            animator.SetBool("IsShoot", false);

        var animatorStateInfo = animator.GetCurrentAnimatorStateInfo(0);

        if (timer >= animatorStateInfo.length && animatorStateInfo.IsName("Shoot"))
        {
            weapon.Shoot();
            timer = 0;
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {

    }
}
```

Листинг программы для скрипта *AttackBehaviourZombie.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class ChaseBehaviorSwat : StateMachineBehaviour
{
    NavMeshAgent agent;
    float attackMinRange = 5;
    float attackMaxRange = 10;
    float chaseRange = 15;

    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent = animator.GetComponent<NavMeshAgent>();
        agent.speed = 4;

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (!pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsShoot", false);
            animator.SetBool("IsChase", false);
            pursuitEnemy.isPursuit = false;
            return;
        }
```

```
        agent.SetDestination(pursuitEnemy.enemy.transform.position);
        float distance = Vector3.Distance(animator.transform.position, pursuitEnemy.enemy.transform.position);

        if (distance <= attackMaxRange)
            animator.SetBool("IsShoot", true);

        if (distance <= attackMinRange || distance > chaseRange)
        {
            pursuitEnemy.isPursuit = false;
            animator.SetBool("IsChase", false);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent.SetDestination(agent.transform.position);
        //agent.speed = 0.5f;
    }
}
```

Листинг программы для скрипта *AttackRunBehaviourSwat.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class AttackRunBehaviorSwat : StateMachineBehaviour
{
    private float attackMinRange = 5;
    private float attackMaxRange = 10;

    private float timer = 0;

    private NavMeshAgent agent;

    private PursuitEnemy pursuitEnemy;

    private GameObject gun;

    private Weapon weapon;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        pursuitEnemy = animator.GetComponent<PursuitEnemy>();

        agent = animator.gameObject.GetComponent<NavMeshAgent>();
        agent.speed = 0.5f;
        weapon = animator.GetComponent<Swat>().weapon;
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer += Time.deltaTime;

        if (!pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsShoot", false);
            animator.SetBool("IsChase", true);
            return;
        }

        animator.transform.LookAt(pursuitEnemy.enemy.transform);
        agent.SetDestination(pursuitEnemy.enemy.transform.position);
        float distance = Vector3.Distance(animator.transform.position, pursuitEnemy.enemy.transform.position);

        if (distance <= attackMinRange)
            animator.SetBool("IsChase", false);

        if (distance > attackMaxRange)
            animator.SetBool("IsShoot", false);

        var animatorStateInfo = animator.GetCurrentAnimatorStateInfo(0);

        if (timer >= animatorStateInfo.length && animatorStateInfo.IsName("WalkAndShoot"))
```

```
        {
            weapon.Shoot();
            timer = 0;
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent.SetDestination(agent.transform.position);
    }
}
```

Листинг программы для *ChaseBehaviourSwat.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class ChaseBehaviorSwat : StateMachineBehaviour
{
    NavMeshAgent agent;
    float attackMinRange = 5;
    float attackMaxRange = 10;
    float chaseRange = 15;

    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent = animator.GetComponent<NavMeshAgent>();
        agent.speed = 4;

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (!pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsShoot", false);
            animator.SetBool("IsChase", false);
            pursuitEnemy.isPursuit = false;
            return;
        }

        agent.SetDestination(pursuitEnemy.enemy.transform.position);
        float distance = Vector3.Distance(animator.transform.position, pursuitEnemy.enemy.transform.position);

        if (distance <= attackMaxRange)
            animator.SetBool("IsShoot", true);

        if (distance <= attackMinRange || distance > chaseRange)
        {
            pursuitEnemy.isPursuit = false;
            animator.SetBool("IsChase", false);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent.SetDestination(agent.transform.position);
        //agent.speed = 0.5f;
    }
}
```

Листинг программы для *ChasebehaviourZombie.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
```

```
public class ChaseBehaviorSwat : StateMachineBehaviour
{
    NavMeshAgent agent;
    float attackMinRange = 5;
    float attackMaxRange = 10;
    float chaseRange = 15;

    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent = animator.GetComponent<NavMeshAgent>();
        agent.speed = 4;

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (!pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsShoot", false);
            animator.SetBool("IsChase", false);
            pursuitEnemy.isPursuit = false;
            return;
        }

        agent.SetDestination(pursuitEnemy.enemy.transform.position);
        float distance = Vector3.Distance(animator.transform.position, pursuitEnemy.enemy.transform.position);

        if (distance <= attackMaxRange)
            animator.SetBool("IsShoot", true);

        if (distance <= attackMinRange || distance > chaseRange)
        {
            pursuitEnemy.isPursuit = false;
            animator.SetBool("IsChase", false);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent.SetDestination(agent.transform.position);
        //agent.speed = 0.5f;
    }
}
```

Листинг программы для скрипта *CheckDetectPlayer.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using static DetectEnemy;

public class CheckDetectPlayer : MonoBehaviour, ICheckDetect
{
    List<IPursuit> pursuits = new List<IPursuit>();

    // Update is called once per frame
    void Update()
    {
        if (pursuits.Count > 0)
            Player.instance.isDetect = CheckDetect();
    }

    bool CheckDetect()
    {
        pursuits.RemoveAll(x => !x.IsPursuit);

        foreach (IPursuit pursuit in pursuits)
        {
            if (pursuit.IsPursuit)
                return true;
        }
```

```
        return false;
    }

    public void IsDetect(string detectName, IPursuit pursuit)
    {
        if (detectName == name)
            pursuits.Add(pursuit);
    }
}
```

Листинг программы для скрипта *DetectEnemy.cs*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class DetectEnemy : MonoBehaviour
{
    [SerializeField]
    private LayerMask objectSelectionMask;

    public float chaseRange = 10;
    public float fireRange = 50;

    public bool isDetect = false;
    public DetectInfo currDetectEnemy;

    public List<string> enemyNames = new List<string>();
    List<DetectInfo> enemies = new List<DetectInfo>();

    // Start is called before the first frame update
    void Start()
    {
        foreach (var name in enemyNames)
        {
            var objs = GameObject.FindGameObjectsWithTag(name);

            if (objs.Length > 0)
            {
                enemies.AddRange(objs.Select(x => new DetectInfo(x, gameObject)).Where(x => x.detectedInfo.IsCanMove));
            }
        }

        if(enemies.Count == 0)
            this.enabled = false;
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        if (FindEnemyNearby(out DetectInfo enemy))
        {
            string name;

            if (enemy.detectedCharacter == null)
                name = enemy.GetCharacter().name;
            else
                name = enemy.detectedCharacter.name;

            if (enemy.detectedInfo.IsChange &&
                !IsEnemyName(name) &&
                enemy.detectedInfo.Magnitude <= 0.5 ||
                name == enemy.discoveredObject.name)
            {
                isDetect = false;
                return;
            }

            if (enemy.Distance < chaseRange && !enemy.detectedInfo.IsHidden)
            {
                Vector3 direction = enemy.detectedObject.transform.position - transform.position;
                Quaternion rotation = Quaternion.LookRotation(direction);
                transform.rotation = Quaternion.Lerp(transform.rotation, rotation, Time.deltaTime * 3);
            }

            Ray ray = new Ray(transform.position + new Vector3(0, 0.5f, 0), transform.forward);
```

```
        Debug.DrawRay(transform.position + new Vector3(0, 0.5f, 0), transform.forward * fireRange);
        if (Physics.Raycast(ray, out RaycastHit hitInfo, fireRange, objectSelectionMask))
        {
            //Debug.Log(hitInfo.collider.name);

            if (name == hitInfo.collider.name)
            {
                isDetect = true;
                currDetectEnemy = enemy;
            }
        }
    }
    else
        isDetect = false;
}

bool IsEnemyName(string name)
{
    return enemyNames.Any(x => x == name);
}

private DetectInfo lastDetect = new DetectInfo();

EnemyComparer enemyComparer = new EnemyComparer();

bool FindEnemyNearby(out DetectInfo enemy)
{
    enemies.Sort(enemyComparer);

    var newDetect = enemies[0];

    if (newDetect.GetDistance() <= chaseRange && newDetect.detectedInfo.IsCanMove && !newDetect.Equals(lastDetect))
    {
        enemy = enemies[0];
        return true;
    }
    else
    {
        enemy = new DetectInfo();
        return false;
    }
}

class EnemyComparer : IComparer<DetectInfo>
{
    public int Compare(DetectInfo p1, DetectInfo p2)
    {
        if (p1.detectedInfo.IsCanMove && !p2.detectedInfo.IsCanMove)
            return -1;
        else if (!p1.detectedInfo.IsCanMove && p2.detectedInfo.IsCanMove)
            return 1;
        else
            return (int) (p1.GetDistance() - p2.GetDistance());
    }
}

public struct DetectInfo
{
    public GameObject detectedObject;

    public GameObject detectedCharacter;

    public Transform discoveredObject;

    public IMove detectedInfo;

    private float distance;

    public float Distance { get => distance; }

    public DetectInfo(GameObject detectedObject, GameObject discoveredObject)
    {
        this.detectedObject = detectedObject;
        this.discoveredObject = discoveredObject.GetComponent<IMove>().Character.transform;
        detectedInfo = detectedObject.GetComponent<IMove>();
        detectedCharacter = detectedInfo.Character;
        distance = 0;
    }
```

```
        public float GetDistance()
        {
            distance = Vector3.Distance(discoveredObject.position, detectedObject.transform.position);
            return distance;
        }

        public GameObject GetCharacter()
        {
            detectedCharacter = detectedInfo.Character;
            return detectedCharacter;
        }

        public override int GetHashCode()
        {
            return HashCode.Combine(detectedObject, detectedCharacter, discoveredObject, detectedInfo, distance);
        }

        public override bool Equals(object obj)
        {
            return obj is DetectInfo info &&
                   EqualityComparer<GameObject>.Default.Equals(detectedObject, info.detectedObject) &&
                   EqualityComparer<GameObject>.Default.Equals(detectedCharacter, info.detectedCharacter) &&
                   EqualityComparer<IMove>.Default.Equals(detectedInfo, info.detectedInfo) &&
                   distance == info.distance;
        }
    }
}
```

Листинг программы для скрипта *GameData.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class GameData
{
    //reference to itself
    public static GameData current;

    //Data members that mirror playerProgress class

    //Game completion bool
    public bool isGameCompleted;

    public bool room1;
    public bool room2;
    public bool room3;
}
```

Листинг программы для скрипта *GameManager.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    public static GameManager instance;

    public bool loadGameFile;

    public bool isGameComplete = false;

    public bool gameEnded = false;

    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
        else if (instance != this)
        {
            Destroy(gameObject);
            Debug.LogWarning("Another instance of GameManager has been created and destoryed!");
```

```
        }

        DontDestroyOnLoad(this);
    }

    public void SetPlayerLocation(Transform sp)
    {
        Player.instance.transform.position = sp.position;
    }

    public void NewGame()
    {
        isGameComplete = false;
        loadGameFile = false;
        gameEnded = false;
    }

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Листинг программы для скрипта *GameProgress.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameProgress : MonoBehaviour
{
    public static GameData gameData;

    //bools to store which level has been completed
    public bool room1 = false;
    public bool room2 = false;
    public bool room3 = false;

    //Game completion bool
    public bool isGameCompleted = false;

    private void Awake()
    {
        //Check if the game needs to load. Set by the main menu
        if (GameManager.instance.loadGameFile)
        {
            LoadGame();
        }
    }

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void SaveGame()
    {
        GameData.current.room1 = room1;
        GameData.current.room2 = room2;
        GameData.current.room3 = room3;

        SaveAndLoad.Save();
    }
```

```
    public void LoadGame()
    {
        SaveAndLoad.Load();

        room1 = GameData.current.room1;
        room2 = GameData.current.room2;
        room3 = GameData.current.room3;
    }
}
```

Листинг программы для скрипта *Health.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Health : MonoBehaviour, IHealth
{
    public float health;
    public float maxHealth;

    public void AddHealth(float count)
    {
        health += count;

        if(health > maxHealth)
            health = maxHealth;
    }

    public void RemoveHealth(float count)
    {
        health -= count;

        if(health < 0)
            health = 0;
    }

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if(health == 0)
            Death();
    }

    private void Death()
    {
        GetComponent<IMove>().IsCanMove = false;
    }
}
```

Листинг программы для скрипта *ICheckDetect.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface ICheckDetect
{
    void IsDetect(string detectName, IPursuit pursuit);
}
```

Листинг программы для скрипта *IdleBehaviourSwat.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IdleBehaviorSwat : StateMachineBehaviour
{
    float timer;
    float timeStoping;
```

```
    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer = 0;
        timeStoping = Random.Range(5, 11);

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer += Time.deltaTime;
        if (timer > timeStoping)
            animator.SetBool("IsPatrolling", true);

        if (pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsChase", true);
            animator.SetBool("IsPatrolling", false);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {

    }
}
```

Листинг программы для скрипта *IdleBehaviourZombie.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IdleBehaviorZombie : StateMachineBehaviour
{
    float timer;
    float timeStoping;

    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer = 0;
        timeStoping = Random.Range(5, 11);

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        timer += Time.deltaTime;
        if (timer > timeStoping)
            animator.SetBool("IsWalk", true);

        if (pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsRunning", true);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {

    }
}
```

Листинг программы для скрипта *IHealth.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IHealth
{
    void AddHealth(float count);
    void RemoveHealth(float count);
}
```

Листинг программы для скрипта *IMove.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IMove
{
    bool IsChange { get; }
    bool IsCanMove { get; set; }
    bool IsHidden { get; set; }
    float CurrSpeed { get; set; }
    float Speed { get; }
    float MaxSpeed { get; }
    float Magnitude { get; }
    bool IsRunning { get; }
    bool IsSneakWalk { get; }
    float Horizontal { get; }
    float Vertical { get; }
    Vector3 Direction { get; set; }
    float RotationSpeed { get; }
    GameObject Character { get; set; }

    void UpdateState();
}
```

Листинг программы для скрипта *IPersuit.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IPursuit
{
    bool IsPursuit { get; }
    IMove EnemyInfo { get; }
}
```

Листинг программы для скрипта *Level.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Level : MonoBehaviour
{
    public Transform spawnPoint;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Листинг программы для скрипта *Loader.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class Loader : MonoBehaviour
{
    //PreFab referneces
    public GameObject player;
    public GameObject manager;


    void Awake()
    {
        //We create the static gameObjects once.
        if (GameManager.instance == null)
        {
            Instantiate(manager);
        }

        if (Player.instance == null)
        {
            Instantiate(player);
        }
    }
}
```

Листинг программы для скрипта *MainMenu.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using System.IO;
using UnityEngine.UI;

public class MaimMenu : MonoBehaviour
{
    public GameObject manager;

    public Button loadGameButton;

    private void Awake()
    {
        if (GameManager.instance == null)
        {
            Instantiate(manager);
        }
    }

    private void Start()
    {


        if (CheckLoadGame())
        {
            loadGameButton.gameObject.SetActive(true);
        }
    }

    public void PlayeGame()
    {
        GameManager.instance.NewGame();
        NextSceneManager.instance.LoadLevelScene("Level1");
    }

    public void ExitGame()
    {
        Debug.Log("Игра закрылась");
        Application.Quit();
    }

    public void RunLoadGame()
    {
        GameManager.instance.loadGameFile = true;
        NextSceneManager.instance.LoadLevelScene("Level1");
    }

    bool CheckLoadGame()
    {
        return File.Exists(Application.persistentDataPath + "/savedGames.gd");
    }
```

```
}
```

Листинг программы для скрипта *MovePlayer.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovePlayer : MonoBehaviour
{
    private Rigidbody _rigidbody;

    public IMove moveState;

    private void Start()
    {
        _rigidbody = Player.instance.body;
        moveState = Player.instance.moveState;
    }

    // Update is called once per frame
    void Update()
    {
        if(moveState.IsCanMove)
            Move();
    }

    private void Move()
    {
        moveState.Direction = transform.TransformDirection(Vector3.ClampMagnitude(new Vector3(moveState.Horizontal, 0,
moveState.Vertical), 1));

        if (moveState.IsRunning &&
            moveState.CurrSpeed <= moveState.MaxSpeed &&
            !moveState.IsSneakWalk)
            moveState.CurrSpeed = moveState.MaxSpeed;
        else
            moveState.CurrSpeed = moveState.Speed;

        _rigidbody.velocity = moveState.Direction * moveState.CurrSpeed;

        if (moveState.IsSneakWalk || moveState.Magnitude < 0.5f || !moveState.IsCanMove)
            moveState.IsHidden = true;
        else
            moveState.IsHidden = false;

        Rotation();
    }

    private void Rotation()
    {
        if (moveState.Magnitude > 0.01)
            moveState.Character.transform.rotation = Quaternion.Lerp(moveState.Character.transform.rotation,
Quaternion.LookRotation(moveState.Direction), Time.deltaTime * moveState.RotationSpeed);
    }
}
```

Листинг программы для скрипта *MoveState.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MoveState : MonoBehaviour, IMove
{
    [SerializeField]
    private bool isCanMove = true;
    public bool IsCanMove { get => isCanMove; set => isCanMove = value; }

    [SerializeField]
    private bool isChange = true;
    public bool IsChange { get => isChange; }

    [SerializeField]
    private bool isHidden = false;
    public bool IsHidden { get => isHidden; set => isHidden = value; }
```

```
    private float currSpeed = 0;
    public float CurrSpeed { get => currSpeed; set => currSpeed = value; }

    [SerializeField]
    private float speed = 2;
    public float Speed { get => speed; }

    [SerializeField]
    private float maxSpeed = 4;
    public float MaxSpeed { get => maxSpeed; }
    public float Magnitude { get => direction.magnitude; }

    [SerializeField]
    private float rotationSpeed = 10;
    public float RotationSpeed { get => rotationSpeed; }

    private Vector3 direction;
    public Vector3 Direction { get => direction; set => direction = value; }

    public bool IsRunning { get; private set; } = false;
    public bool IsSneakWalk { get; private set; } = false;
    public float Horizontal { get; private set; } = 0;
    public float Vertical { get; private set; } = 0;

    [SerializeField]
    private GameObject character;
    public GameObject Character { get => character; set => character = value; }

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void UpdateState()
    {
        IsRunning = Input.GetKey(KeyCode.LeftShift);
        IsSneakWalk = Input.GetKey(KeyCode.C);
        Horizontal = Input.GetAxis("Horizontal");
        Vertical = Input.GetAxis("Vertical");
    }
}
```

Листинг программы для скрипта *MoveWithMouse.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MoveWithMouse : MonoBehaviour
{
    private float xRot;
    private float yRot;
    public Camera _camera;
    public float sensivity = 10f;
    private Rigidbody _rigidbody;
    private IMove moveState;

    // Start is called before the first frame update
    void Start()
    {
        _rigidbody = Player.instance.body;
        moveState = Player.instance.moveState;
    }

    // Update is called once per frame
    void Update()
    {
        xRot += Input.GetAxis("Mouse X");
        yRot += Input.GetAxis("Mouse Y");

        _camera.transform.rotation = Quaternion.Lerp(_camera.transform.rotation, Quaternion.Euler(-yRot, xRot, 0), Time.deltaTime * 10);
```

```
        if(moveState.IsCanMove)
            _rigidbody.rotation = Quaternion.Euler(0, xRot, 0);
    }
}
```

Листинг программы для скрипта *NextSceneManager.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class NextSceneManager : MonoBehaviour
{
	public static NextSceneManager instance = null;

	void Awake()
	{

		//Set the instance only once.
		if (instance == null)
		{
			instance = this;
		}
		else if (instance != this)
		{
			//Enforces that there will always be one instance of a gameObject. This is for type errors prevention
			Destroy(gameObject);
			Debug.LogWarning("Another instance of NextSceneManager have been created and destoryed!");
		}

	}
	//Load / unloads a level scene with fade or not
	public void LoadLevelScene(string sceneName, Animator fader = null, Image im = null)
	{
		if (!SceneManager.GetSceneByName(sceneName).isLoaded)
		{
			// will allow a fade if the needed vars are not null
			if (im != null && fader != null)
			{
				if (!fader.GetBool("Fade"))  // Really make sure we don't keep calling the coroutine while the scene
is still the fade transition.
				{
					//Fade to next scene
					StartCoroutine(FadeToNextScene(fader, im, sceneName));
				}
			}
			else
			{
				//Load new Scene
				SceneManager.LoadScene(sceneName, LoadSceneMode.Single);
			}
		}
	}

	//Will start a wait for fade transition. It will wait once the fade had fully faded to load in the nextr scene
	IEnumerator FadeToNextScene(Animator f, Image i, string sceneName)
	{
		f.SetBool("Fade", true);
		// wait until fade has ended
		yield return new WaitUntil(() => i.color.a >= 1);
		//Load next scene
		SceneManager.LoadScene(sceneName, LoadSceneMode.Single);

	}
}
```

Листинг программы для скрипта *PatrolBehaviour.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
```

```
public class NextSceneManager : MonoBehaviour
{
	public static NextSceneManager instance = null;

	void Awake()
	{

		//Set the instance only once.
		if (instance == null)
		{
			instance = this;
		}
		else if (instance != this)
		{
			//Enforces that there will always be one instance of a gameObject. This is for type errors prevention
			Destroy(gameObject);
			Debug.LogWarning("Another instance of NextSceneManager have been created and destoryed!");
		}

	}
	//Load / unloads a level scene with fade or not
	public void LoadLevelScene(string sceneName, Animator fader = null, Image im = null)
	{
		if (!SceneManager.GetSceneByName(sceneName).isLoaded)
		{
			// will allow a fade if the needed vars are not null
			if (im != null && fader != null)
			{
				if (!fader.GetBool("Fade"))  // Really make sure we don't keep calling the coroutine while the scene
is still the fade transition.
				{
					//Fade to next scene
					StartCoroutine(FadeToNextScene(fader, im, sceneName));
				}
			}
			else
			{
				//Load new Scene
				SceneManager.LoadScene(sceneName, LoadSceneMode.Single);
			}
		}
	}

	//Will start a wait for fade transition. It will wait once the fade had fully faded to load in the nextr scene
	IEnumerator FadeToNextScene(Animator f, Image i, string sceneName)
	{
		f.SetBool("Fade", true);
		// wait until fade has ended
		yield return new WaitUntil(() => i.color.a >= 1);
		//Load next scene
		SceneManager.LoadScene(sceneName, LoadSceneMode.Single);

	}
}
```

Листинг программы для скрипта *PauseMenu.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class PauseMenu : MonoBehaviour
{
   //bool for checking if the game has been paused on other scripts
   public static bool isGamePaused = false;
   //menu sceen reference
   public GameObject pauseMenu;
   //hacking tool to be shown when in puzzle and it is puased
   public GameObject hackingTool;
   //Audio sliders UI refs
   public Slider musicSlider;
   public Slider sFXSlider;
   public Slider puzzleSFXSlider;
   //a bool to check if the audio sliders has been set with the proper values
   private bool setSliderListen = false;


   void Update()
```

```
    {
        //check if the player has pressed on the cancel key
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (isGamePaused)
            {
                ResumeGame();
            }
            else
            {
                PauseGame();
            }
        }
    }

    public void PauseGame()
    {
        isGamePaused = true;
        //Freeze time
        Time.timeScale = 0;
        //show pause screen
        pauseMenu.SetActive(true);
        //if (!NextSceneManager.instance.isPuzzleLoaded)
        //{
        //    hackingTool.SetActive(false);
        //}
        //first time audio slider sets
        if (!setSliderListen)
        {
            setSliderListen = true;

            //musicSlider.value = AudioMix.instance.musiclvl;
            //sFXSlider.value = AudioMix.instance.SFXlvl;
            //puzzleSFXSlider.value = AudioMix.instance.puzzlvl;

            //musicSlider.onValueChanged.AddListener(delegate { AudioMix.instance.SetMusicLvl(musicSlider.value); });
            //sFXSlider.onValueChanged.AddListener(delegate { AudioMix.instance.SetSFXLvl(sFXSlider.value); });
            //puzzleSFXSlider.onValueChanged.AddListener(delegate { AudioMix.instance.SetPuzzLvl(puzzleSFXSlider.value); });
        }
    }

    public void ResumeGame()
    {
        isGamePaused = false;
        //Unfreeze time
        Time.timeScale = 1f;
        //unshow the pause screen
        pauseMenu.SetActive(false);
        //hackingTool.SetActive(true);
    }
    //Trigger a back to the main menu procedure
    public void Menu()
    {

        NextSceneManager.instance.LoadLevelScene("Menu");
        ResumeGame();
    }
}
```

Листинг программы для скрипта *Player.cs*:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    public static Player instance = null;

    [HideInInspector]
    public IMove moveState;
    [HideInInspector]
    public Rigidbody body;
    [HideInInspector]
    public Animator animator;
    [HideInInspector]
    public AnimationPlayer animationPlayer;
    [HideInInspector]
    public bool isDetect = false;
```

```
    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
        else if (instance != this)
        {
            //Enforces that there will always be one instance of a gameObject. This is for type errors prevention
            Destroy(gameObject);
            Debug.LogWarning("Another instance of Player have been created and destoryed!");
        }

        moveState = GetComponent<IMove>();
        body = GetComponent<Rigidbody>();
        animator = GetComponent<Animator>();
        animationPlayer = GetComponent<AnimationPlayer>();
    }

    public void SetCharacter(GameObject gameObject, Avatar avatar)
    {
        moveState.Character = gameObject;
        animationPlayer.SetAvatar(avatar);
    }

    // Update is called once per frame
    void Update()
    {
        moveState.UpdateState();
    }
}
```

Листинг программы для скрипта *PatrolBehaviourZombie.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class PotrolBehaviorZombie : StateMachineBehaviour
{
    NavMeshAgent agent;

    [SerializeField]
    private float movementSpeed;

    [SerializeField]
    private float moveDistance = 40f;

    private PursuitEnemy pursuitEnemy;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent = animator.gameObject.GetComponent<NavMeshAgent>();
        agent.speed = movementSpeed;
        agent.SetDestination(RandomNavSphere(moveDistance, animator.transform.position));

        pursuitEnemy = animator.GetComponent<PursuitEnemy>();
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (agent.remainingDistance <= agent.stoppingDistance)
        {
            animator.SetBool("IsWalk", false);
        }

        animator.SetFloat("Speed", agent.velocity.magnitude / agent.speed);

        if (pursuitEnemy.isPursuit)
        {
            animator.SetBool("IsRunning", true);
        }
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
```

```
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {

    }

    private Vector3 RandomNavSphere(float distance, Vector3 position)
    {
        Vector3 randomDirection = Random.insideUnitSphere * distance;

        randomDirection += position;

        NavMeshHit navHit;

        NavMesh.SamplePosition(randomDirection, out navHit, distance, -1);

        return navHit.position;
    }
}
```

Листинг программы для скрипта *PulseColor.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PulseColor : MonoBehaviour
{
    public GameObject lightsObj;
    List<Light> lights = new List<Light>();

    public Color fromColor;
    public Color toColor;

    Color tmp;

    public float speed = 10;

    float time = 0;

    // Start is called before the first frame update
    void Start()
    {
        foreach (Light light in lightsObj.GetComponentsInChildren<Light>())
            lights.Add(light);
    }

    // Update is called once per frame
    void Update()
    {
        time += Time.deltaTime;

        if (lights[0].color == toColor)
        {
            tmp = toColor;
            toColor = fromColor;
            fromColor = tmp;
            time = 0;
        }

        foreach (Light light in lights)
            light.color = Color.Lerp(fromColor, toColor, Mathf.Abs(Mathf.Sin(time * speed)));
    }
}
```

Листинг программы для скрипта *PursuitEnemy.cs*:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using static DetectEnemy;

public class PursuitEnemy : MonoBehaviour, IPursuit
{
    public bool isPursuit = false;
    public GameObject enemy;
    IMove enemyInfo;
```

```
    DetectEnemy detectEnemy;
    string oldTagCharacter;
    public bool isChangeCharacter = false;

    public bool IsPursuit { get => isPursuit; }

    public IMove EnemyInfo { get => enemyInfo; }

    public Action<string, IPursuit> onAction;

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

    // Start is called before the first frame update
    void Start()
    {
        detectEnemy = GetComponent<DetectEnemy>();
        foreach (GameObject checkDetect in checkDetects)
            onAction += checkDetect.GetComponent<ICheckDetect>().IsDetect;
    }

    // Update is called once per frame
    void Update()
    {
        if (!isPursuit && detectEnemy.isDetect)
        {
            enemy = detectEnemy.currDetectEnemy.detectedObject;
            enemyInfo = detectEnemy.currDetectEnemy.detectedInfo;
            oldTagCharacter = detectEnemy.currDetectEnemy.detectedCharacter.tag;
            onAction(detectEnemy.currDetectEnemy.detectedObject.name, this);
            isPursuit = true;
        }
        else if (enemyInfo != null && enemyInfo.Character.name != oldTagCharacter && isChangeCharacter)
        {
            isPursuit = false;
        }

        if (enemyInfo != null && !enemyInfo.IsCanMove)
        {
            isPursuit = false;
        }
    }
}
```

Листинг программы для скрипта *Reincarnation.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Reincarnation : MonoBehaviour
{
    public GameObject effect;
    public GameObject defaultCharacter;
    public Avatar defaultAvatar;
    private IMove moveState;

    // Start is called before the first frame update
    void Start()
    {
        moveState = Player.instance.moveState;
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.R) && moveState.IsCanMove)
            ChangeCharacter(defaultCharacter);
    }

    private void OnTriggerStay(Collider other)
    {
        if (Input.GetKeyDown(KeyCode.E))
        {
            ChangeCharacter(other.gameObject);
        }
    }

    public void ChangeCharacter(GameObject gameObject)
```

```
    {
        Destroy(moveState.Character);

        moveState.Character = Instantiate(gameObject, transform.position, transform.rotation);
        moveState.Character.name = moveState.Character.name.Replace("(Clone)", "");
        moveState.Character.transform.SetParent(transform);

        if(gameObject.transform.parent != null)
            Player.instance.SetCharacter(moveState.Character, gameObject.GetComponentInParent<Animator>().avatar);
        else
            Player.instance.SetCharacter(moveState.Character, defaultAvatar);

        Instantiate(effect, new Vector3(transform.position.x, transform.position.y + 1, transform.position.z), Quaternion.identity);
    }
}
```

Листинг программы для скрипта *SaveAndLoad.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

public static class SaveAndLoad
{
    //Save method.
    public static void Save()
    {
        //sets binary formatter
        BinaryFormatter bf = new BinaryFormatter();
        //opens the filestream and saves file to local appdata
        FileStream file = File.Create(Application.persistentDataPath + "/savedGames.gd");
        //serilalizes the gamedata
        bf.Serialize(file, GameData.current);
        //closes the file
        file.Close();
    }
    //Save method.
    //public static void SaveIdentification()
    //{
    //    //sets binary formatter
    //    BinaryFormatter bf = new BinaryFormatter();
    //    //opens the filestream and saves file to local appdata
    //    FileStream file = File.Create(Application.persistentDataPath + "/IdentificationData.gd");
    //    //serilalizes the gamedata
    //    bf.Serialize(file, IdentifyData.current);
    //    //closes the file
    //    file.Close();
    //}

    //Load method
    public static void Load()
    {
        //check if file exists
        if (File.Exists(Application.persistentDataPath + "/savedGames.gd"))
        {
            //set the binary formatter
            BinaryFormatter bf = new BinaryFormatter();
            //open the filestream
            FileStream file = File.Open(Application.persistentDataPath + "/savedGames.gd", FileMode.Open);
            //assign the current game data
            GameData.current = (GameData)bf.Deserialize(file);
            //close the file
            file.Close();
        }
    }

    //Load method. Returns false if the file does not exists
    //public static bool LoadIdentification()
    //{
    //    //check if file exists
    //    if (File.Exists(Application.persistentDataPath + "/IdentificationData.gd"))
    //    {
    //        //set the binary formatter
    //        BinaryFormatter bf = new BinaryFormatter();
    //        //open the filestream
    //        FileStream file = File.Open(Application.persistentDataPath + "/IdentificationData.gd", FileMode.Open);
```

```
    //      //assign the current game data
    //      IdentifyData.current = (IdentifyData)bf.Deserialize(file);
    //      //close the file
    //      file.Close();
    //  }
    //  else
    //  {
    //      return false;
    //  }
    //  return true;
    //}
}
```

Листинг программы для скрипта *Swat.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class Swat : MonoBehaviour
{
    public Weapon weapon;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Листинг программы для скрипта *UIHelper.cs*:

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class UIHelper : MonoBehaviour
{
    public GameObject character;

    public GameObject hide;
    public GameObject show;

    [SerializeField]
    private LayerMask objectSelectionMask;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Debug.DrawRay(Player.instance.transform.position + Vector3.up, Player.instance.transform.forward * 1.5f);
        if (Physics.Raycast(Player.instance.transform.position + Vector3.up, Player.instance.transform.forward, out RaycastHit hit, 1.5f,
objectSelectionMask))
        {
            character.SetActive(true);
            character.GetComponent<RectTransform>().position = hit.transform.position + Vector3.up + Vector3.right;
        }
        else if(character.activeSelf)
            character.SetActive(false);

        if (Player.instance.isDetect)
        {
            hide.SetActive(false);
```

```
            show.SetActive(true);
        }
        else
        {
            hide.SetActive(true);
            show.SetActive(false);
        }
    }
}
```

Листинг программы для скрипта *Weapon.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Weapon : MonoBehaviour
{
    public float damage = 5;
    public float fireRate = 1;
    public float range = 50;
    public GameObject muzzleFlash;
    public AudioClip shotSFX;
    public AudioSource _audioSource;

    public Transform bulletSpawn;
    public GameObject hitEffect;

    [SerializeField]
    private LayerMask objectSelectionMask;

    public float force = 150f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void Shoot()
    {
        _audioSource.PlayOneShot(shotSFX);
        Instantiate(muzzleFlash, bulletSpawn.position, bulletSpawn.rotation);

        if (Physics.Raycast(transform.position, transform.right, out RaycastHit hitInfo, range, objectSelectionMask))
        {
            GameObject impact = Instantiate(hitEffect, hitInfo.point, Quaternion.LookRotation(hitInfo.normal));

            Destroy(impact, 2);

            //Debug.Log(hitInfo.collider.name);
            hitInfo.collider.GetComponentInParent<IHealth>().RemoveHealth(5);
            hitInfo.collider.GetComponentInParent<Rigidbody>().AddForce(-hitInfo.normal * force, ForceMode.Impulse);
        }
    }
}
```

Листинг программы для скрипта *ZombieAI.cs*:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class ZombieAI : MonoBehaviour
{
    private NavMeshAgent _navMeshAgent;
    private Animator _animator;

    [SerializeField]
    private float movementSpeed;
```

```
    [SerializeField]
    private float maxChangePositionTime = 10f;

    [SerializeField]
    private float moveDistance = 20f;

    private void Start()
    {
        _navMeshAgent = GetComponent<NavMeshAgent>();
        _navMeshAgent.speed = movementSpeed;
        _animator = GetComponent<Animator>();
        InvokeRepeating(nameof(MoveAnimal), 0, Random.Range(0, maxChangePositionTime));
    }

    private void Update()
    {
        _animator.SetFloat("Speed", _navMeshAgent.velocity.magnitude / movementSpeed);
    }

    private Vector3 RandomNavSphere(float distance)
    {
        Vector3 randomDirection = Random.insideUnitSphere * distance;

        randomDirection += transform.position;

        NavMeshHit navHit;

        NavMesh.SamplePosition(randomDirection, out navHit, distance, -1);

        return navHit.position;
    }

    private void MoveAnimal()
    {
        _navMeshAgent.SetDestination(RandomNavSphere(moveDistance));
    }
}
```