**Отчет по экзаминационной работе**

**Кобрин Максим Андреевич**

**Введение.**

Моими целями в рамках проекта Самосбор(до этого МорЛафкрафт) было написание кода на языке c# для создания различных механик в среде разработки юнити и повышение знаний по ООП и программированию в целом .В более узком смысле моими целями была реализация таких механик как общее взаимодействие игрока с игровыми объектами, системы диалогов с вариантами выбора, системы квестов, системы сохранений, системы здоровья и механики смерти персонажа, создание тумана через и написание скрипта нанесения урона для этого тумана. Кроме того я работал над всякими мелкими задачами по типу: создание для

Дверей, переключателей света и так далее. Также как лидер небольшой группы в рамках большой группы самосбора, я помогал с реализацией различных механик моему подопечному в рамках нашего проекта. А именно:

Система передвижения(работал над бегом и приседанием, система выносливости игрока, главное меню и меню паузы).

Полный список скриптов в рамках моей работы над проектом можно посмотреть здесь: <https://github.com/Zolotushka1/MorLovecraft/tree/cospero/Assets/Scripts/cospero> - ссылка на мою ветку проекта

Технологическая часть

* 1. Система сохранений.   
     Сохранения были реализованы мной через бинарную сериализацию. То есть представления различных объектов Unity в виде файла, содержащего двоичный код. Такой вид сериализации был выбран из-за необходимости сохранения большого количества объектов на сцене(объекты для взаимодействия, предметы, которые можно подобрать в инвентарь).

* + 1. PlayerDataSaveLoad

SavePlayer

При нажатии в игре кнопки сохранения в этом скрипте вызывается SavePlayer, который передает в BinarySavingSystem 3 экземпляра класса: Player\_Move - Игрок(сохранение позиции персонажа), Stats – здоровье игрока и InventoryManager – состояние инвентаря.

LoadPlayer

При нажатии кнопки загрузки в PlayerDataSaveLoad вызывается LoadPlayer, который сначала получает из скрипта BinarySavingSystem десиреализованные *data* из файла сохранения ([ССЫЛКА](https://ibb.co/28R8qY7) смотреть примечания к диаграмме). После получения необходимых для загрузки сохранения данных на сцене происходит поиск классов, которые нужно загрузить, поиск производится через FindObjectOfType <Название класса >();. Когда объект найден, к примеру это Player\_Move, происходит принудительная инициализация необходимого для загрузки класса(1.2 *– это ссылка на абзац*). Таким образом мы как бы запускаем в нужных нам скриптах функцию Start, назначая все переменные, которые нужны для нормальной работы скрипта и выполнения загрузки игры. Это нужно для того, чтобы избежать ошибки, когда при загрузке сохранения, например из гл меню, мы пытаемся обратиться к скрипту, в котором еще не успела запустится функция Start(). Подобную процедуру мы проделываем для всех загружаемых объектов, в которых загрузка завязана на изменении значений в скриптах.

* + 1. BinarySavingSystem **–**

Это класс, предназначенный для бинарной сериализации и десериализации данных.

SavePlayer

Метод SavePlayer() принимает экземпляры классов Stats, Player\_Move и InventoryManager, создает новый объект PlayerData, используя эти экземпляры классов, и сохраняет его на диск в бинарном формате по пути Application.persistentDataPath + "/player.b";.

LoadPlayer

Метод LoadPlayer() загружает сохраненные данные игрока с диска, используя бинарный десериализатор. Он проверяет наличие сохраненного файла в указанном пути и возвращает объект PlayerData.

Если файл не найден, скрипт печатает в консоль строчку Debug.LogError("Save file not found in " + path); где path путь, по которому должен находится файл сохранения.

* + 1. PlayerData

Необходимые для создания сохранения данные об объектах собираются в классе PlayerData. На данном этапе развития проекта собираются следующие параметры: из скрипта передвижения Player\_move - position(в виде набора из 3 флоатов), здоровье из скрипта Stats в виде переменной типа float и состояние инвентаря. Инвентарь сохраняется ввиде 2 списков, количесвто элементов которых, совпадает с числом ячеей инвентаря. Списки называются ItemNames и ItemAmounts, имена предмета и количества предметов соответственно. То есть для каждой ячейки по индексу, равному номеру этой ячейки, в списках сохраняются эти 2 значения. Если ячейка пустая, то она остается пустой в файле сохранения.

* + 1. LoadPromise –  
       вспомогательный скрипт, который позволяет загружать сохранение из главного меню или сцены, отличной от той, которая загружается. При загрузке сохранения таким образом, скрипт создает в загружаемой сцене DontDestroyOnLoad объект, который при пробуждении запускает LoadPlayer из PlayerDataSaveLoad. После этого объект удаляется из сцены.
    2. Подключаемые модули:  
       1) System.Runtime.Serialization.Formatters.Binary; - позволяет сариализовать данные и сохранять их в файл.   
       2) TMPro; - позволяет в коде изменять TextMeshPro объекы текста

3) Ink.Runtime; - позволяет взаимодействовать с историями из .INK.   
создавать, перемещатся по ним и выводить их в виде текста на экран.

* 1. Диалоговая система. Система диалогов была реализована через файлы формата .ink и через библиотеку INK в c#. В ink файлах прописаны ветки диалогов в виде графа описанного текстом вида: (тут будет пример кода в инке). На языке c# Разработан объект DialogueManager. Он отвечает за работу с историей инка. История инка(Story) это объект формата ink, который содержит описание всех возможных вариантов реплик игрока и ответов npc, связанных между собой причинно-следственной связью. Например: если выбрать реплику A, то возникнут варианты выбора из блока C(будет пример). В самом коде DialogueManager происходит подключение определенного ink файла и его открытие, отображение вариантов ответа на сцене Unity и функции которые позволяют выбирать реплики, путем клика на них в диалоговом окне. Кроме того код динамически создает кнопки(варианты ответов в диалоге, в зависимости от количества этих вариантов ответа). Возможности диалоговой системы(здесь будет фото окна диалогов и варианты развития диалогов).
  2. Общая система взаимодействия – позволяет игроку взаимодействовать с различными игровыми объектами. То есть вызывать различные функции на определенных объектах, находящихся на слое “Cursor” при нажатии кнопки взаимодействия на клавиатуре.
     1. UICursor – скрипт, определяющий возможно ли взаимодействие с объектом, находящимся в поле зрения игрока. Проверка происходит следующим образом: методом Update() постоянно рисуется невидимый луч длины Distanse(Этот параметр регулируется в инспекторе и показывает, насколько далеко должен быть игрок, для взаимодействия с объектами). Этот луч попадая в какой-либо коллайдер, проверяет имеет ли этот коллайдер тег “Cursor”, если имеет, то через info.collider. GetComponent проверяет наличие компонента Activator на объекте. При выполнении обоих условий, взаимодействие становится возможным и при нажатии игроком соответствующей клавиши, у меня это “E”, происходит выполнение методов, назначенных под компонентом Activator.
     2. Activator – это класс, задачей которого является выполнение любых методов, присвоенных в инспекторе UnityEvent-у OnActivatedButtonDown и проигрывание аудиоклипов, если они назначены в том же инспекторе в компоненте Activator. OnActivatedButtonDown – часть класса Activaror, и вызав на нем метода .Invoke() как раз запускает все методы, на которые данный объект подписан. 1.3 - это ссылка на абзац

1.4 Туман и нанесение урона. Туман в нашей игре перекрывает некоторые области уровня и, распространяясь по сцене, заставляет игрока действовать быстрее. Я реализовал туман через партиклы, то есть частицы. Сам объект состоит из партиклов и сферического коллайдера. Когда игрок попадает в него, запускается функция OnColliderStay(), в нем вызывается метод GetDamage() из скрипта Stats().

* 1. 1 Распространение тумана. Распространения тумана происходит через активацию заранее выставленных на сцену префабов тумана. Каждый туман знает, какой туман заспавнить следующим, и при своей активации запускает таймер на спавн следущего тумана.

# РЕАЛИЗУЕМАЯ архитектура

* 1. Изображение архитектуры системы сохранения [ССЫЛКА](https://ibb.co/28R8qY7) – хостинг картинок

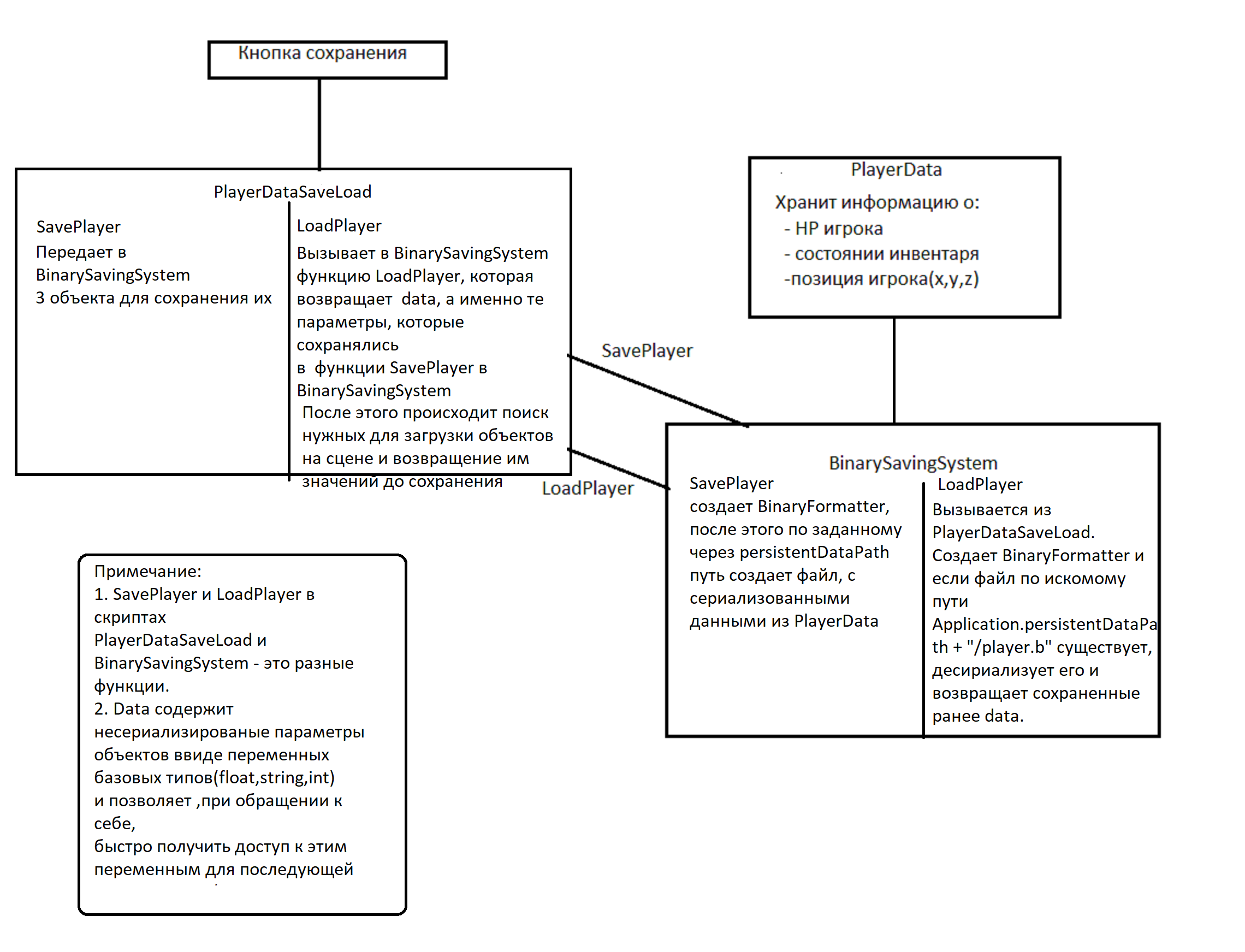


Рис 1. диаграмма работы системы сохранения

* 1. Инициализация объектов при загрузке сохранения, пример из кода

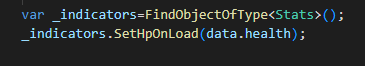


Рис 2. Запуск функции инициализации в скрипте PlayerDataSaveLoad

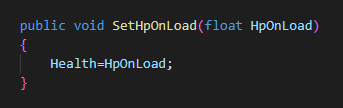
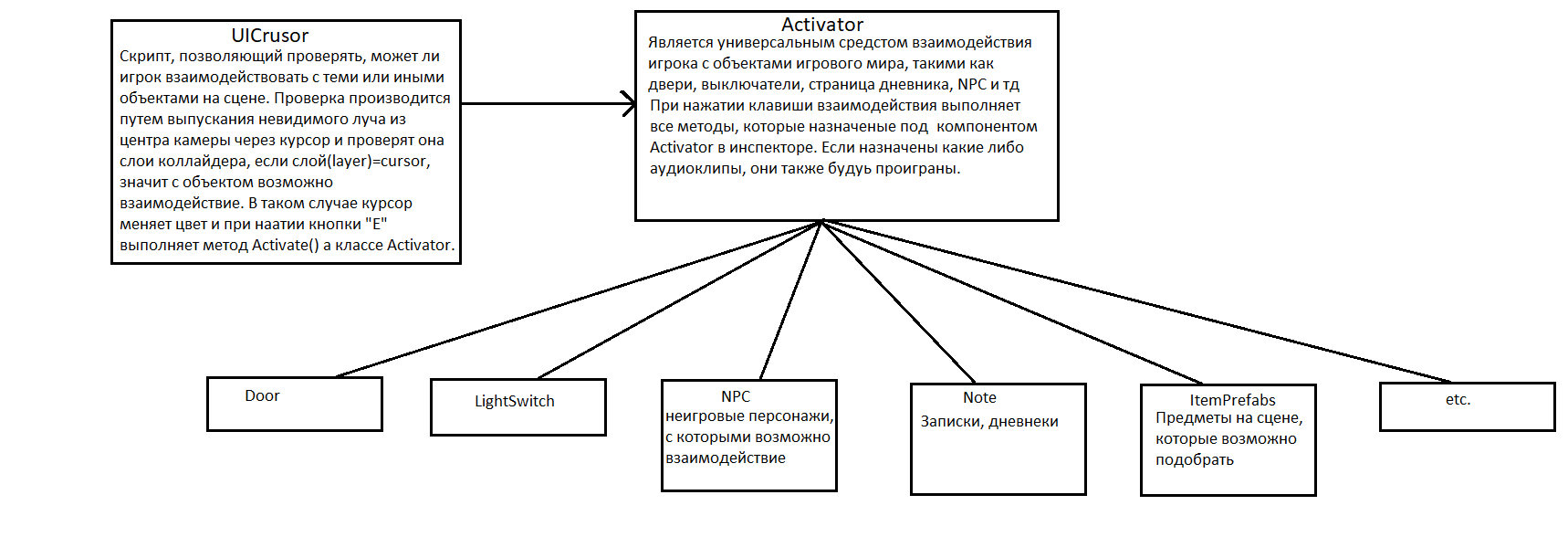


Рис 3. Это сама функция, производящая выключение CharacterController для изменения положения игрока. По завершении замены позиции контроллер снова включается.

* 1. Диаграмма системы взаимодействия. [ссылка](https://ibb.co/vmWdLFx) - хостинг картинок

В

Выводы

Почти все задачи проекта, которые мне ставил лидер команды или же я сам были реализованы в том виде, в котором я их себе представлял. Мной не была полностью реализована диалогов и квестов. Т.К, на данном этапе проекта эти задачи не являются первостепенными и было принято согласованное с лидером решение отложить их на потом. То же касается и системы сохранений, но на нее просто не хватило времени. В планах у меня создание системы сохранений со списком сейвов и соответственно со множеством файлов сохранений и возможность загрузки любого из этих вайлов в любой момент. Кроме того необходимо осуществить сохранение и загрузку объектов на сцене при создании и загрузке файла сохранения , например квестовые предметы или предметы, которые можно подобрать в инвентарь. Работа над данным проектом расширила мои знания ООП и программирования в целом, дала понимание о работе в среде разработки Unity и создании игр.