Министерство образования и науки РФ

ФГАОУ ВПО «Северо-Восточный федеральный университет имени М.К. Аммосова»

Институт математики и информатики

Кафедра информационных технологий

Мекумянов Александр Геннадьевич

Выпускная квалификационная работа

по специальности 010400«Информационные технологии»

**Разработка конструктора двумерных игр**

**на платформе PSM**

Допущен к защите Научный руководитель:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2014 г. Павлов А. В., к.ф.-м.н., доцент кафедры ИТ

Зав. кафедрой, д.т.н.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Мордовской С. Д./

Якутск – 2014

Содержание

[Введение 4](#_Toc390975322)

[1. Платформа PlayStation Mobile и средства разработки для нее 6](#_Toc390975323)

[1.1 Рынок игровых устройств 6](#_Toc390975324)

[1.2 PlayStation Vita 7](#_Toc390975325)

[1.3 Платформа PlayStation Mobile 10](#_Toc390975326)

[1.4 PSMSDK 11](#_Toc390975327)

[1.5 Обзор разработки приложений для PlayStationMobile 12](#_Toc390975328)

[1.6 Игры в жанре платформер 15](#_Toc390975329)

[1.6.1 Компьютерные игры 15](#_Toc390975330)

[1.6.2 Разработка двумерных компьютерных игр 17](#_Toc390975331)

[1.6.3 Игра Piercenfield 18](#_Toc390975332)

[1.6.4 Требования к платформеру 18](#_Toc390975333)

[1.7. Определение требований к конструктору двумерных игр 19](#_Toc390975334)

[2.1.1 Конструкторы игр 19](#_Toc390975335)

[2.1.2 Существующие конструкторы и их особенности 20](#_Toc390975336)

[2.1.3 Требования к конструктору и разработка архитектуры 23](#_Toc390975337)

[2. Разработка конструктора двумерных игр для платформы PSM 25](#_Toc390975338)

[2.1. Архитектура конструктора 25](#_Toc390975339)

[2.2. Игровой движок 27](#_Toc390975340)

[2.3.1 Разбор игровых движков 27](#_Toc390975341)

[2.3.2 Существующие игровые движки и их особенности 28](#_Toc390975342)

[2.3.3 Требования игрового движка 31](#_Toc390975343)

[2.3. Разработка игрового движка 31](#_Toc390975344)

[2.3.1 Разработка компоненты графики 31](#_Toc390975345)

[2.3.2 Разработка класса AnimationSprite 34](#_Toc390975346)

[2.3.3 Разработка компоненты физики 38](#_Toc390975347)

[2.3.4 Разработка класса Game Physics Engine для физики 43](#_Toc390975348)

[2.3.5 Разработка стандартных объектов движка 45](#_Toc390975349)

[2.3.6 Разработка движка навигации между игровыми сценами 54](#_Toc390975350)

[2.3.7 Интеграция скриптового языка Lua 60](#_Toc390975351)

[2.4. Разработка интерфейса конструктора 70](#_Toc390975352)

[2.4.1 Разработка интерфейса 70](#_Toc390975353)

[2.4.2 Интеграция GLEED2D к интерфейсу 71](#_Toc390975354)

[2.4.3 Интеграция игрового движка и интерфейса 80](#_Toc390975355)

[2.5. Финальные тесты 80](#_Toc390975356)

[Заключение 84](#_Toc390975357)

[Список использованной литературы 86](#_Toc390975358)

# Введение

Коллектив молодых энтузиастов, в который я вхожу, разрабатывает двумерную игру Piercenfield для консоли PlayStationVita. При разработке игры возникли проблемы. Объекты игры расставлялись программно прямо в коде игры, так что при изменении координат приходилось переком­пи­ли­ровать проект. Дизайнеры могли работать с игрой только через програм­миста, а программисты разрабатывали движок и не могли заниматься дизайном игровых сцен. Скрипты — программные компоненты, отвечающие за логику игры, — записывались прямо в код, из-за чего он становился объемным и нечитабельным. Игру планировалось отправить на конкурс под названием Independent Game Festival (Фестиваль не зависимых игр). Из-за отсутствия хорошего стабильного игрового движка, а также отсутствия графического редактора, с которым могли бы работать дизайнеры, игра не успела к конкурсу.

Результатом опыта разработки сложной игры стало осознание необходимости ее полной переделки. Необходимо было вывести скрипт за пределы кода, создать графический редактор игровых сцен и строго разделить дизайн и программирование. Во время переделки игры было решено сделать конструктор для создания игр.

**Целью** данной выпускной квалификационной работы является разработка конструктора двумерных игр на платформе PSM, используя возможности PSM SDK.

**Актуальность** данной темы определяется тем, что PlayStationVITA является одной из самых молодых консолей на данный момент. Рынок игр для нее на данный момент не насыщен, что означает меньший уровень конкуренции и большие возможности для независимых разработчиков. Для платформы PSM открытых и бесплатных конструкторов игр нет. Конст­руктор игры позволит начинающим разработчикам игр облегчить создание игр, избежав шаблонных повторяющихся строк кода. Для этого достаточно знать базовый уровень программирования.

**Задачи работы.** Для достижения цели работы были поставлены следую­щие задачи:

1. Изучить существующие конструкторы и разработать архитектуру для собственного конструктора
2. Изучить существующие игровые движки
3. Разработать игровой движок, его архитектуру и реализовать его компоненты
4. Разработать интерфейс конструктора
5. Объединить интерфейс и игровой движок, получив единую систему конструктора
6. Протестировать полученный код

Работа состоит из введения, двух глав и заключения. В первой главе обзор платформы PSM и процесса разработки двумерных игр для нее. Глава завершается определением требований к конструктору. Во второй главе подробно рассмотрена разработка конструктора двумерных игр в соответ­ствии с этими требованиями. Список литературы содержит 17 наименований.

# 1. Платформа PlayStation Mobile и средства разработки для нее

## 1.1 Рынок игровых устройств

Игровая приставка — это специализированное электронное устройство, разработанное и созданное для видеоигр. На данный момент доминирующее положение на рынке игровых приставок занимают консоли 7-го поколения, это Sony PlayStation 3, XBOX 360 и NintendoWii. Также зимой прошлого года стартовала продажа приставок 8-го поколения(Sony PlayStation 4, XBOX One).

В таблице ниже приведены объемы продаж наиболее распространенных игровых приставок:

|  |  |  |
| --- | --- | --- |
| **Консоль** | **Продано** | **Датавыхода** |
| Nintendo Wii | 99,84 млн (на 31.03.2013)[1] | 19.11.2006 |
| Microsoft Xbox 360 | 75,9 млн (на 31.12. 2012) [2] | 22.11.2005 |
| SonyPlayStation 3 | около70млн (на04.11.2013) [3] | 11.11.2006 |

Существует подвид игровых приставок — портативные (карманные) игровые приставки. Такие системы имеют собственное встроенное устройство отображения (ни к чему не приставляются), поэтому называть их игровыми приставками несколько некорректно.

Одной из успешных портативных консолей является PlayStation Portable от компании Sony. Запуск консоли начался успешно, с более чем 200 тыс. проданных приставок в первый день продаж. В настоящее время для нее вышло более 800 игр.

Существует угроза для портативных устройств со стороны мобильных/планшетных устройств, так как их покупает большинство людей. Согласно данным исследования аналитического агентства ABIResearch, в первом квартале текущего года на рынок было отгружено 405 млн. мобильных телефонов. Из них 197 млн. составили смартфоны. Лидером по динамике роста в течение двух лет являются мобильные игры, что объясняется активным развитием этого сегмента.

Главным конкурентом Sony в области домашних игровых консолей является корпорация Microsoft со своим брендом Xbox. 14 марта 2006 года Microsoft представила XNA – набор инструментов с управляемой средой времени выполнения (.NET), облегчающий разработку и управление компьютерными играми. С помощью XNA можно делать игры под Windows PC, Xbox 360 и Zune[4].

Игры для Xbox 360, написанные в XNA Game Studio могут быть представлены Creators Club community, для чего требуется членство, стоящее 49 долларов США за 4 месяца или 99 долл. в год. Все игры, представленные сообществу, подвергаются равной оценке других разработчиков. Если игра успешно проходит рецензирование, она попадает на Xbox Live Marketplace. Таким образом, Microsoft поддерживает независимых разработчиков. Однако имеются опасения, что с выходом Xbox One(8-е поколение консолей) Microsoft прекратит поддержку сообщества.

Sony в последнее время начала поддерживать независимых разработчиков, позволив им разрабатывать игры для своей новой портативной консоли PlayStation Vita.

## 1.2 PlayStation Vita

**PlayStation Vita (PSV)** — портативная консоль от Sony Computer Entertainment, являющаяся преемником PlayStation Portable.Выпущена в Японии 17 декабря 2011 года, в Северной Америке и Европе — 22 февраля 2012 года.

Система была анонсирована 27 января 2011. До анонса консоль была известна как PSP2, или Next Generation Portable (NGP). Одновременно с консолью была представлена платформа PlayStation Suite (PSM), затем переименованная в Playstation Mobile. В России консоль впервые представили на ИгроМире-2011.

**Технические характеристики:**

* Центральный процессор: 4-ядерный процессор ARMCortex-A9 (частотой до 2.0 GHz, может быть снижена Sony для экономии заряда батареи).
* Графический процессор: Power VRSGX543MP4+ с поддержкой OpenGL 2.0 (200 MHz, 133 MPolygon/s, 4GPixel/s) и объемом видеопамяти 128 мегабайт.
* Оперативная память: 512 мегабайт.
* Дисплей: 5-дюймовый (127мм) SUPER AMOLED Plus от SAMSUNG-дисплей, 960 × 544 (16:9) 220 dpi, TrueColor.
* Размеры (ширина × высота × толщина): приблизительно 182×83,55×18,6 мм.
* Средства ввода: кнопка PS, кнопка питания, крестовина (вверх/вниз/вправо/влево), кнопки действий (https://lh3.googleusercontent.com/MVllmPkETjILLtIhPQZbZaJMKDb8bx3-ZY78CO6vsWVZYzex1n2pUEKYFMvtJsFtCcdHLJp2uIgLLw0Suhjox9EeZtOXgzA-G5tVs3Ns1_dFTHxyDpzsTDk5uw, https://lh3.googleusercontent.com/0t-txFjCZW7CyT3vuG-cwPXlyiYgsf0NpYzn0dSWIo_8P49vHEuJvkLaorvNMJdZJrH7FD49DY6qE6QOHiltzp5QdibvMZ-olzic97v7cahz2PfGWB9C9mMtSA, https://lh5.googleusercontent.com/zYVpbtilcAg891-_x7z_MiTI8zHE7zs1lvew6ahSS9Ipfk8YhQ2-eY9F08iij9RjdsBlMsD6Ycd-ygbcvhuaJ546lqVOz7fseR2q_u6A7akz3UCmA6w3eIcBHQ, https://lh5.googleusercontent.com/dzq3iETobdlt7f9joLQNSpFGpiXM0VprWMZx5oyXF9ZZIJR0KtvkcffqtTMLt5JzbQEIvIO80hMmptfZ_6sGXxOyC7fliKjdCwEwb-4DH4AeI4mfZhZYYFvWyQ), боковые кнопки (L, R), 2 аналоговых стика, кнопки Start и Select, кнопки регулировки громкости (+/-); также емкостные сенсорный экран и задняя сенсорная панель с multi-touch.
* Коммуникации: адаптеры IEEE 802.11 b/g/n Wi-Fi и Bluetooth 4.0 + EDR, а также опциональный 3G-модем.
* Камеры: передняя и задняя, запись с частотой кадров 120 кадров/сек при 320x240 и 60 кадров/сек при 640 × 480.
* Дополнительно: стереодинамики, микрофон, трехосный электронный компас, трехосевые гироскоп и акселерометр; также GPS в модели с 3G.
* Поддержка RemotePlay (использование консоли в качестве контроллера для PlayStation 3 по сети).

В PlayStation Vita вместо XMB реализован новый интерфейс, одна из особенностей которого — окно Live Area, которое есть у каждого приложения. В нём находится последняя информация об активности ваших друзей, электронная инструкция и кнопка запуска приложения. В целом интерфейс представляет собой центральное вертикально перелистываемое меню с круглыми иконками игр и приложений.



Рис 1. Внешний вид PlayStationVita.

В начале августа 2011 года компания *Sony* бесплатно предоставила некоторым независимым разработчикам пакет разработчика (devkit) бесплатно, а в середине августа того же года объявлена официальная цена на *devkit* для разработки игр для *PSVita* — 1 900 евро, в то время как аналогичные комплекты для *PlayStation 2* и *PlayStation 3* стоили по 20 000 евро, а для PSP — 15 000 евро.

DevKit позволяет разрабатывать программы, непосредственно обращающиеся к аппаратуре PSVita (native apps). С другой стороны, в ноябре 2011 года на конференции GDC 2011 были рассказаны подробности о PlayStation Suite, позже переименованном в PlayStation Mobile — комплекте средств разработки для создания приложений и игр для PSVita, XperiaPlay, Tablet, TabletP и других будущих устройств, предназначенном для независимых разработчиков. Разработка этих приложений осуществляется на языке C#, который работает в адаптированной компанией Sony виртуальной машине на основе фреймворка Mono.[16] Средой разработки служит PSM Studio, которая в свою очередь тоже является адаптированной версией IDE Mono Develop.

## 1.3 Платформа PlayStation Mobile

PlayStation Mobile является ответом Sony на вызовы рынку игр со стороны смартфонов и планшетов. Компания пытается привлечь мобильных пользователей и разработчиков относительно несложных игр с тем чтобы, с одной стороны, повысить привлекательность своих устройств за счет расширения каталога доступных для них игр, а с другой стороны, «заманить» пользователей с более демократичного рынка смартфонов/планшетов и игр для них на более прибыльный рынок консолей.

На данный момент онлайн-сервис PlayStation Mobile, предлагающий загрузку приложений, совместимых одновременно с сертифицированными смартфонами и планшетов и консолями PSVita, запущен лишь в некоторых странах. Однако Sony прикладывают все усилия, чтобы распространить программу на другие регионы. Первым шагом в этом нелегком деле стал релиз официального пакета разработчика, который теперь доступен всем желающим абсолютно бесплатно. А для публикации в PlaystationStore уже надо будет заплатить $100 в год.

Для получения доступа к инструментарию необходимо зарегистрироваться и скачать его на [официальном сайте](http://li.ru/go?https://psm.playstation.net/). Там же доступен обширный F.A.Q. с наиболее актуальными вопросами, возникающими у начинающих разработчиков.

Сара Томпсон (Sarah Thomson), менеджер Sony по мобильному контенту, считает, что запуск PlayStation Mobile серьезно увеличит количество приложений и игр, которые будут доступы на сертифицированных смартфонах и планшетах, а также консолях PSVita: «Мы убираем барьеры на пути начинающих разработчиков к созданию и публикации собственных проектов, мы сделали процесс более легким и приятным».

Уже сейчас в PSMobile в ряде стран доступно множество игр от независимых разработчиков по достаточно демократичным ценам, сравнимым с AppStore и GooglePlay. С развитием сервиса количество опубликованных приложений должно серьезно возрасти и пополниться не только развлекательным софтом, но и чем-то более необходимым.

## 1.4 PSMSDK

PlayStation Mobile SDK (комплект разработки программного обеспечения) является средой разработки приложений, которые могут выполняться на нескольких платформах.

Используя PSM SDK, можно написать приложение, которое будучи скомпилированным, даст один двоичный файл, который может быть запущен на PSVita, Android и других устройствах.

Таким образом, разработчику не требуется писать отдельный код для каждого устройства, и эффективность разработки приложений может быть значительно повышена.

В настоящее время для большинства Android-устройств среда исполнения PSM-программ должны быть загружена с сайта Sony, в дальнейшем планируется сделать ее предустанавливаемой.

PSM SDK предлагает эмулятор на основе Windows для запуска PSM-приложений, таким образом, прототип можно разработать, отладить и проверить на эмуляторе, прежде чем запускать его на других платформах.



**Рис 2.** Некоторые устройства с поддержкой PSM

PSM поддерживается на следующих устройствах:

**Sony:** PlayStationVita, SonyTabletP и S, большинство моделей смартфонов Xperiaна базе ОС Android: A SO-04ENEW, acro HD IS12S/SO-03D, acro IS11S/SO-02C, acro S LT26w, arc SO-01C, AX SO-01E, GX SO-04D, ion LT28at/LT28i/LT28h, L C2104/C2105, NX SO-02D, PLAY R800i/R800a/R800at/R800x/SO-01D, S LT26i, SL LT26ii, SP C5302/C5303/C5306, SX SO-05D, T LT30a/LT30p, TL LT30at, TX LT29i, UL SOL22, V LT25i, VL SOL21, Z C6602/C6603/SO-02E, ZL C65s02/C6503/C6506, ZR C5502/C5503, а так же XperiaTablet Z SO-03E/SGP311/SGP312/SGP321/SGP331;

**Sharp:** AQUOS PHONE EX SH-04E,AQUOS PHONE SERIE ISW16SH, AQUOS PHONE SERIE SHL21,AQUOS PHONE Xx SoftBank 106SH, AQUOS PHONE Xx SoftBank 203SH, AQUOS PHONE ZETA SH-02E, AQUOS PHONE ZETA SH-09D, PANTONE® 6 Soft Bank 200SH, AQUOS PAD SHT21;

**HTC:** One EVO 4G LTE, One S,One SVNEW, One V, One X, One XL,One X+;

**Fujitsu:** ARROWS Tab F-05E;

**Wikipad:** Wikipad 7.

**ПК Windows:**(В эмуляторе, необходимо наличие видеокарты с поддержкой OpenGL 3.0 или позднее)

## 1.5 Обзор разработки приложений для PlayStationMobile

Приложение платформы PSM представляет собой байт-код для адаптированной машины Mono. Mono—это проект по созданию полноценной реализации системы .NET Framework на базе свободного программного обеспечения. Имеются версии Mono для ОС Linux, Windows, MacOSX, iOS и др. Microsoft прекратила поддержку своего набора инструментов для разработки игр XNA. Проект MonoGame поставил целью независимую реализацию версии 4 платформы XNA. Имеются реализации для Windows, Linux, WindowsPhone 8, iOS.[4]

Исходный код компилируется в управляемом коде (PSM приложения) с помощью компилятора C#. На рисунке показана структура среды исполнения [5].

C:\Diplom2\ИТ-10 Иванов Б\2012-Jordan Game Summit-PlayStation Mobile.emf

**Рис 1.** Структура среды исполнения

Есть собственная среда разработки на языке программирования С#, PSM Studio. XNA версии 1.0 тоже имел собственную среду разработки (IDE). Сейчас имеется версия XNA 4.0, которая представляет собой SDK, совместимый с Visual Studio 2010. Sony также указывает в документации к PSM Studio, что в дальнейшем планирует интеграцию с Visual Studio.

Созданное PSM приложение компилируется для каждой платформы. Eдиный бинарник, исполняемый: внутри среды управления Mono эмулятора на PC/среды управления PSM на Android/ на PSVita.

Для загрузки контента путь указывается относительно папки Application. Папка Applicationнаходится в папке Debug/Release.

**Архитектура приложения**

Проект Playstation Mobile Application состоит обычно из трех частей: Инициализация(Initialization), Обновление(Update), Рисовка(Render) и имеется главная функция Main, в которой две последние вызываются в бесконечном цикле.

public static void Main (string[] args)

{

**Initialize ();**

while (true)

{

SystemEvents.CheckEvents ();

**Update ();**

**Render ();**

}

}

Это типичная архитектура каркаса игры. Такой каркас используют в DirectX[6], XNA[4] и т.д.

Теперь подробнее остановимся на каждой части.

1. public static void **Initialization**()

Обычно в этой функции загружают контент и инициализируют объекты. В XNA, например эта функция разделена на две: Intitalization и LoadContent.

public static void Initialize ()

{

// Set up the graphics system

graphics = new GraphicsContext ();

}

2. public void static **Update**()

В этой части программы будет вычисляться вся логика игры. Например вычисление координаты, коллизии, нажатий клавиш и т. д. Крайне не рекомендуется (невозможно) рисовать объекты.

public static void Update ()

{

// Query gamepad for current state

var gamePadData = GamePad.GetData (0);

}

3. public void static **Render**()

Здесь собственно происходит отрисовка.

public static void Render ()

{

// Clear the screen

graphics.SetClearColor (0.0f, 0.0f, 0.0f, 0.0f);

graphics.Clear ();

// Present the screen

graphics.SwapBuffers ();

}

## 1.6 Игры в жанре платформер

### 1.6.1 Компьютерные игры

**Компьютерная игра** (иногда используется неоднозначный термин **видеоигра**) — компьютерная программа, служащая для организации игрового процесса, связи с партнёрами по игре, или сама выступающая в качестве партнера.

Компьютерные игры часто создаются на основе фильмов и книг; есть и обратные случаи. С 2011 года компьютерные игры официально признаны в США отдельным видом искусства.

Компьютерные игры оказали столь существенное влияние на общество, что в информационных технологиях отмечена устойчивая тенденция к геймификации для неигрового прикладного программного обеспечения.[7]

Компьютерные игры состоят из:

* Сеттинг
* Геймплей
* Музыки

Сеттинг — это среда, в которой происходит действие художественного произведения, настольной или компьютерной игры; место, время и условия действия. Одна игра может выходить в разных сеттингах. Несмотря на то, что сеттинг — это в первую очередь художественное оформление игры, он опосредованно влияет и на её правила: например, в фэнтэзийной игре регламентируется применение магии и описываются различные фэнтезийные расы со своими достоинствами и недостатками (люди, эльфы, хоббиты, орки, гоблины и т. д.).

Геймплей — игровой процесс компьютерной игры с точки зрения игрока. Геймплей включает в себя разные аспекты компьютерной игры, в том числе технические, такие как внутри игровая механика, совокупность определенных методов взаимодействия игры с игроком и др. Само понятие геймплея крайне обобщено и обычно используется для выражения полученных ощущений в ходе прохождения игры, под влиянием таких факторов, как графика, звук и сюжет.

Основными компонентами игрового процесса компьютерных игр являются:

* Цель, которой должен достичь игрок.
* Препятствие будь то враги(Мобы) или загадки, которые будут мешать игроку на пути к цели.
* Очки собирая которых игроку будет легче избавляться от врагов.

Компьютерные игры в основном классифицируются по жанрам, а также по количеству игроков. Вследствие того, что критерии принадлежности игры к тому или иному жанру не определены однозначно, классификация компьютерных игр недостаточно систематизирована, и в разных источниках данные о жанре конкретного проекта могут различаться.

**Платформер**— это жанр компьютерных игр, основной чертой игрового процесса которых является прыгание по платформам, лазанье по лестницам, собирание предметов, обычно необходимых для завершения уровня.[8] Как правило, это игры двумерные.

Основной характеристикой 2Dигры является то, что объекты расположены в двумерной плоскости. Объектами игры могут быть: главный герой, враги, предметы, дверь, бочка, фон и т.д.

### 1.6.2 Разработка двумерных компьютерных игр

Разработкой компьютерных игр занимается разработчик, который может быть представлен как одним человеком, так и фирмой. Обычно крупномасштабные коммерческие игры разрабатываются командами разработчиков в пределах компании, специализирующейся на играх для персонального компьютера или консолей.

Самым важным компонентом, который надо разработать является *и****гровая сцена*. Сцена**— это контейнер уровня/игры. Игровые сцены разрабатывает ведущий дизайнер игры(гейм-дизайнер). Он, как режиссер в спектакле, собирает и контролирует все действия происходящие на сцене. Сцена заполняется персонажами(спрайтами), фоном, действиями и всем на этой сцене заправляет геймплей.[9]

**После расстановки всех объектов на сцене, за дело берется художник(графический дизайнер). Художник начинает рисовать графику, так называемые спрайты. *Спрайт* (sprite)** — это любое изображение на игровой сцене. Спрайты бывают анимированные (Персонажи) и не анимированные (Карта, Деревья, Препятствия и т.п.)[9]

**Все вышеописанное не имеет смысла, если нет Движка. Движок(игровой движок)** — совокупность функций и методов, надстройка над графической подсистемой, позволяющая минимизировать временные затраты на написание игры, поднять разработку на более высокий уровень. Игровой движок не включает в себя геймплей, он скорее позволяет сконцентрироваться на нем, не тратя время на низкоуровневую работу с графикой, звуком и физикой игры [9]. Разрабатывают игровой движок программисты.

Для описания логики объектов и сценария игровых сцен, используются сценарии, или скрипты — программы на интерпретируемых высокоуровневых языках, описывающие действия, выполняемых системой. Если игровой движок имеет поддержку скриптов, то гейм-дизайнеры могут корректировать механику и баланс игры самостоятельно.

### 1.6.3 Игра Piercenfield

Игра разрабатывается командой молодых энтузиастов с целью выхода в рынок и получения бесценного опыта. Сеттинг игры происходит в средние века, где присущие такие вещи как магия и монстры. Геймплей представляет собой прыгание по платформам, уничтожение монстров и развитие главного героя.

Piercenfield разрабатывается исключительно под PlayStationVita (возможно, будет и андроид-версия). Целевой аудиторией являются любители хорошего, вдумчивого и разнообразного боя, глубокого сюжета, а также просто ценители игр от независимых разработчиков (инди-игры), вообще люди всех возрастов.

При разработке игры возникали некоторые проблемы. Объекты игры расставлялись прямо в коде игры, так что при изменении координат приходилось перекомпилировать проект. Дизайнеры могли работать с игрой только через программиста, а программисты разрабатывали движок и не могли заниматься с дизайном игровых сцен. Скрипты игры записывались прямо в код, из-за чего он становился объемным и нечитабельным.

Результатом опыта разработки игры такого масштаба стало осознание необходимости полной переделки игры. Должен быть осуществлен вывод скрипта за пределы кода, создание графического редактора игровых сцен и строго разделить дизайн и программирование.

В то же время было понято, что игра такого жанра строится по достаточно жесткой схеме, допускающей значительную степень автоматизации. Поэтому было решено сделать конструктор.

### 1.6.4 Требования к платформеру

Простой компьютерной игре в жанре платформер требуется реализовать:

* Главное меню игры
* Уровни(сцены) игры, а также переход между ними.
* Спрайтовая анимация, чтобы показать что игрок бегает, ударяет, прыгает и т.д.
* Физика, для приближения прыжков и коллизий к реальности.
* Сценарные (скриптовые) события.
* Врагов и Искусственный Интеллект(ИИ) к ним.
* Музыка и звуки

Будущий конструктор должен помочь со следующими проблемами. Автоматизация уровней, убрав из кода координаты объектов и пути к файлам текстур. Создание Врагов и ИИ. Это объясняется тем, что дизайнеры могли бы сами корректировать механику ИИ без помощи программиста. Избавлению от скриптовых событий внутри кода.

## 1.7. Определение требований к конструктору двумерных игр

В этом разделе рассматриваются существующие конструкторы, выделяются их сильные и слабые стороны, и на основе этого рассмотрения определяются требования к разрабатываемому конструктору двумерных игр.

### 2.1.1 Конструкторы игр

Конструктор игр — программа для быстрой конвейерной сборки компьютерных игр. Объединяет в себе игровой движок и интегрированную среду разработки (IDE), и значительно упрощает процесс создания игр, делая создание игр доступным любителям-непрограммистам. То есть такая программа позволяет создавать игры людям-неспециалистам в области создания компьютерных игр.

Иногда конструкторы игр ограничены тем или иным жанром (например конструкторы: аркад, RPG, или квестов). В других случаях конструкторы позволяют создавать игры различных жанров (предоставляя разработчику большие возможности для программирования), но, возможно, ограничивающая разработчика двумерными играми.

Хорошими примерами игр, сделанных на конструкторе, являются:

1. **Dustforce**

Игра-платформер, разработанная и выпущенная студией HitboxTeam. Игра была выпущена в январе 2012 года дляMicrosoft Windows (доступно в Steam), а в мае 2012 года вышла версия для MacOSX.Первоначально игра была сделана с помощью конструктора GameMaker и была представлена на третьем конкурсе независимых разработчиков компьютерных игр, проводимый порталом IndiePub, в котором заняла первое место и получила главный приз - $100.000.Сейчас продается в Steam.[11]

1. **Wasteland 2**

Пост апокалипсическая ролевая игра от Брайана Фарго и InXileEntertainment. Она будет продолжением игры 1988 годаWasteland (игра Fallout является её духовным преемником) и идейным продолжением серии Fallout. Финансирование игры на 100 % обеспечено фанатами серии и будущими игроками с помощью сервисов Kickstarter и Paypal. За 42 дня проект собрал свыше 3.000.000$. [12]

1. **«Dreamfall Chapters: The Longest Journey»**

Компьютерная игра в жанре квест, продолжение культовых квестов «The Longest Journey»и «Dreamfall: The Longest Journey».«Dreamfall Chapters» представляет собой трёхмерную игру с «point-and-click» интерфейсом и упором на взаимодействие с персонажами, исследование игрового мира и решение головоломок. Работа над игрой была профинансирована фанатами серии, пожертвовавшими более 1.500.000$ через сайт Kickstarter, и Норвежским институтом кино.[17]

### 2.1.2 Существующие конструкторы и их особенности

В ходе разработки конструктора, были рассмотрены и изучены следующие конструкторы компьютерных игр.

1. **GameMaker.**

**GameMaker: Studio** (ранее известный как **GameMaker**) — один из самых известных конструкторов игр. Написан на Delphi. Доступен для ОС Windows, 7-я версия программы также существовала в версии для Mac. Ведущий разработчик — Марк Овермарс. Система рассчитана в основном на создание двухмерных (2D) игр любых жанров. Также подойдёт для создания различных презентаций и т. п. Начиная с 6-й версии появилась ограниченная возможность работать с 3D.Может быть рекомендован для изучения программирования. Будучи профессором утрехтского университета Марк Овермарс начал разрабатывать GameMaker как учебное пособие для своих студентов. GameMaker распространяется на условиях Shareware, бесплатная версия ограничена в функциональности, а при запуске откомпилированных в ней игр показывается логотип программы.

Имеет собственный редактор спрайтов. Все спрайты он хранит в специальной папке под названием Sprites.

Объекты, которые будут лежать в папке Objects, описывается с помощью событий. События состоят из действий и разделяются на несколько типов: циклическое, каллизийное, на нажатие клавиш, и т.д.

Действия, от которых состоит события, имеют типы с которыми нужно работать. Например: при нажатии клавиш меняется направление спрайта, или при коллизии с пулей от которых сокращается жизнь. Создав несколько таких событий получится логика какого-нибудь объекта.

Для создания сцен используется собственный графический редактор на которых вставляются объекты.

Достоинства:

* Кроссплатформенность;
* Интеграция со Steam;
* Хорошая система составлений логик объектов
* Имеет собственные редакторы

Недостатки:

* Плохо оптимизирован для больших игр
* Разработчики перестали развивать режим 3D вообще
* В последнее время GameMaker стал больше предназначен для мобильных платформ.
* Использование собственного языка подразумевает дополнительное время для его изучения.

1. **Unity**

Unity – это профессиональный мультиплатформенный игровой 3D-движок и интегрированный мощный игровой редактор для облегчения создания игр. Поэтому из-за наличия редактора здесь и далее было решено рассмотреть его как конструктор и движок.

Как конструктор он имеет очень удобный и подробный интерфейс с графическим редактором.

Так же как и GameMaker умеет создавать новые объекты игры. Но в отличии от GameMaker каждый объект состоит из так называемых компонент. Компонент могут имеют вид как свойства объекта(каллизийное тело, с выводом графики и т.д.), а также в виде эффектов, утилит, скриптов и т.д. Утилиты используется для описание логики, которые обрабатывают ту или иную логику свойств объекта. Например, если объект является графическим и он должен выводить анимацию, исходя из каких-нибудь факторов, то используют утилиту animator.

Имеет интегрированные утилиты для разработки игры. Например так же как GameMaker имеет редактор спрайтов. Есть также очень мощная утилита обрабатывающая анимации, под названием animator. Она работает по принципу вывода анимации исходя из состояний, в которых объект может быть.

Например может, также как и GameMaker, создавать и хранить у себя сцены.

Скрипты к объектам Unityможно писать на нескольких языках программирования: Javascript, C# и BooScript.

Достоинства:

* Скрипты можно писать на языках программирования
* Наличие утилит нужные для разработки игры
* Не имеет ограничений для дизайна папок проекта

Недостатки:

* Имеет те же недостатки что и у всех конструкторов. Проблемы со скоростью во время интеграции с движком.

### 2.1.3 Требования к конструктору и разработка архитектуры

После рассмотрения существующих конструкторов и изучения их сильных и слабых сторон, были поставлены требования к конструктору. Так, в разобранных конструкторах есть свой графический редактор, поэтому было решено добавить графический редактор. Дабы не ограничивать пользователя в программировании логики, было решено не разрабатывать систему составления логики как в GameMaker'е. Вместо этого было поставлено требование интегрировать произвольный скрипт, подобно Unity, т.е. без ограничений. Было замечено, что все конструкторы имеют возможность создавать объекты с помощью мыши. Поэтому было решено разработать утилиту, создающую мобов и спрайтов. Интерфейс должен использовать принцип Drag&Drop, как у разобранных конструкторов. Не должно быть ограничений при дизайне архитектуры проекта.

Таким образом, простейший конструктор должен включать в себя:

1. Игровой движок, работающий на платформе PSM.
2. Среда разработки с графическим интерфейсом.
3. Собственный язык программирования
4. Инструментарий, необходимый при разработке игры
   * Графический редактор уровней
   * Редактор объектов
   * Редактор скриптов
5. Утилита, создающая мобов и спрайтов

Конструктор должен уметь создавать конвейерную сборку игр. Функциональная спецификация конструктора следующая:

* Сохранить/Загрузить игровой проект
* Создание игровых сцен
* Создание игровых объектов
* Создание и привязка к сценам или объектов скрипта.

# 2. Разработка конструктора двумерных игр для платформы PSM

## 2.1. Архитектура конструктора

**Архитектура конструктора**

Все конструкторы, рассмотренные в предыдущем подразделе, могут компилировать сделанную игру в один исполняемый файл. Разрабатываемый конструктор этого не сможет по ряду причин:

1. PSMStudio, в котором пишется движок, не способен строить исполняемый файл Windows. Вместо этого игра запускается на эмуляторе, который входит в комплект PSMSDK.
2. Обратный инжиниринг процесса сборки "exe"файла, который запускается на эмуляторе, строго запрещен лицензионным соглашением.[13] Сам код эмулятора тоже является закрытым.
3. Во время начала разработки конструктора в Россиии других странах использование PSM Studio было ограничено. Было запрещено запускать приложения на аппаратных устройствах (в телефонах, Playstation VITA и т.д.), а также невозможно было публиковать приложения в магазинах Sony. Ограничение было снято совсем недавно, 29 мая 2014 года.

Из-за перечисленных проблем было решено вынести логику приложений, создающихся в конструкторе, за пределы исполняемого файла. Таким образом, конструктор будет поставляться с заранее собранным исполняемым файлом игры. Этот исполняемый файл будет содержать движок, способный получать настройки и логику из внешних файлов, своих для каждого проекта.

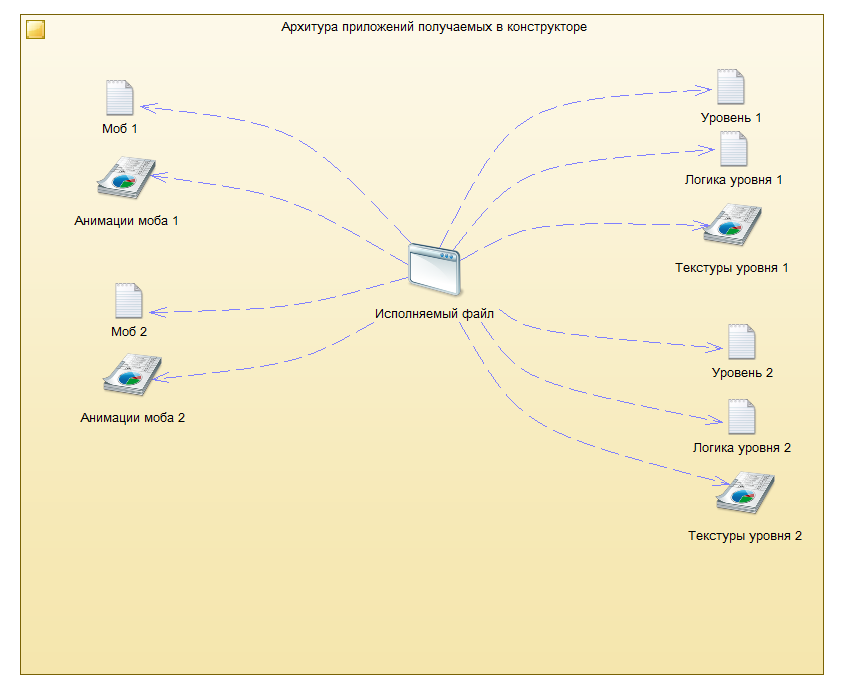


Рис. 1. Архитектура приложений, получаемых в конструкторе.

В соответствии с этим конструктор был разделен на две части:

1. Интерфейс конструктора. Здесь будут реализованы функции добавления уровней, мобов, скриптов и т.д.
2. Игровой движок. Исполняемый файл для эмулятора PSM Studio. Является движком, который должен обрабатывать файлы созданные интерфейсом конструктора.

Интерфейс будет порождать проект-файл, где будут указаны где расположены файлы уровней, скриптов, объектов и т.д, а также GameInit.script главный скрипт запускающий саму игру и где будет указан имя проект файла(рис 2).

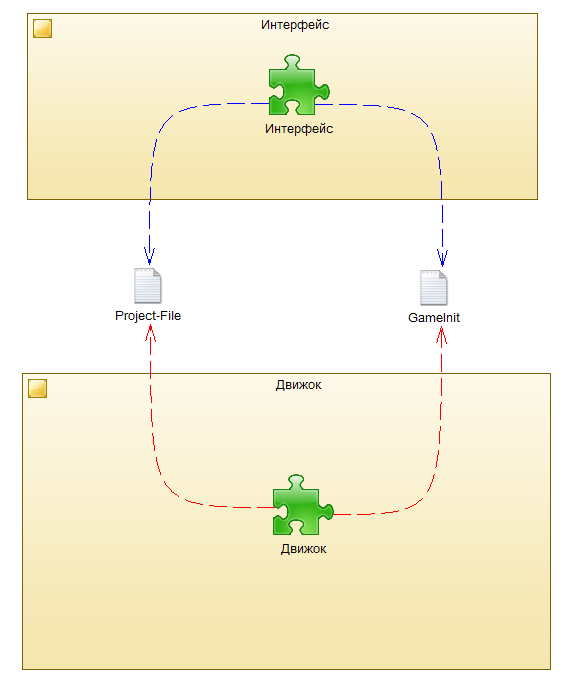


Рис. 2. Архитектура конструктора

Игровой движок же будет парсить данные файлы и на основе этого запустит игру. Логика объектов и сценарии уровня будут реализованы в виде скриптов. При каждом запуске исполняемого файла будет происходить загрузка всего контента и скриптов.

## 2.2. Игровой движок

### 2.3.1 Разбор игровых движков

**Игровой движок** — это центральный программный компонент компьютерных видеоигр или других интерактивных приложений с графикой, обрабатываемой в реальном времени. Он обеспечивает основные технологии, упрощает разработку и часто даёт игре возможность запускаться на нескольких платформах, таких как игровые консоли и настольные операционные системы, например, GNU/Linux, Mac OS X и Microsoft Windows.

Основную функциональность обычно обеспечивает

* игровой движок
* включающий движок рендеринга («визуализатор»),
* физический движок
* звук
* систему скриптов
* анимацию
* искусственный интеллект
* сетевой код
* управление памятью
* многопоточность.

Игровой движок, используя выше указанные компоненты создает функционал игры. Возможность создание мобов и их интеллекта, создание уровней, навигации между сценами и т.д. То есть с помощью игрового движка создается скелет игры.

Часто на процессе разработки можно сэкономить за счёт повторного использования одного игрового движка для создания множества различных игр.

### 2.3.2 Существующие игровые движки и их особенности

Для изучения и просмотра были выбраны следующие движки

1. Unity
2. M.U.G.E.N
3. PopCap Framework

Данные игровые движки являются бесплатными и довольно популярными среди игровых разработчиков.

**Unity**

В подразделе 2.1.2 Unityбыл уже рассмотрен, но в качестве конструктора. Unity изначально был продуман как игровой движок. В этом можно убедиться открыв проект Unityчерез Visual Studio 2010 или MonoDevelop. Движок Unity называется UnityEngine и подключается в проекте. Исходный код движка для бесплатной версии закрыт. Имеет очень богатый функционал, такие как вывод GUI, поддержка устройств ввода и вывода и т.д.

Откроется проект на языке, которая была выбрана при создании скрипта. В данном проекте находятся все добавленные скрипты, которые можно там и редактировать. При описании логики объекта используется стандартный архитектура используемая при разработке игр(см 1.5), за исключением того что вывод графики в скриптах не используется.

Для вычислений коллизий используется свободный и открытый физический движок Box2D.

В адрес Unityимеется очень много критики насчет реализации методов предоставляемые движком(например GUI и в поисковых запросах). Не стоит забывать что Unityв первую очередь предназначена для 3D игр.

**M.U.G.E.N**

M.U.G.E.N — бесплатный (условно-бесплатный для расширенного функционала) графический 2D движок виртуальных сражений, разработанный компанией Elecbyte. Написан на языке программирования Си с использованием библиотек Allegro. Бета версия MUGEN работает под управлением DOS, GNU/Linux и Microsoft Windows. Дата выпуска 17 июля 1999 года. [14]

В основном предназначен для игр в жанре аркада, в котором персонажи выбираются для сражения в случайном порядке, тем не менее, определённый порядок выбора можно задать при помощи сценария.

Представляется в виде конвейерной сборки. Программирование почти не требуется. Проект (Исполняемый файл) позволяет любому создавать своих персонажей, фон и другие объекты игры через интерпретируемые текстовые файлы, графику и звуковые сборки (компиляции). Имеет множество утилит для создания вышеперечисленных объектов. Архитектура M.U.G.E.N'а легла в основе будущего конструктора, в частности работы исполняемого файла с внешними файлами.

**PopCap Framework**

Является ярким представителем классических игровых движков.

Была разработана PopCapGames. PopCapGames — американская компания-разработчик и издатель казуальных компьютерных игр. Расположена в Сиэтле, штат Вашингтон, США. Основана в 2000 году Джоном Вечи, Брайаном Файти и Джейсоном Капалкой. На данный момент в этой компании работает более 400 человек. Основной формой распространения игр компании является цифровая дистрибуция по модели «попробуй и купи», позволяющая попробовать игру перед покупкой.

Компания опубликовала в свободном доступе используемый в её играх движок под названием PopCapGames Framework (также известен как Sexy AppFramework). Движок представляет собой набор программных библиотек, написаных на языке C++, позволяющий быстро разрабатывать игры «в стиле PopCap» для Microsoft Windows. Он является частью программы по сотрудничеству с внешними разработчиками, предлагающей им услуги по изданию игр.

В решении (Solution) подключается как отдельный проект. Требует первоначальной настройки и установки необходимых библиотек таких как, DirectX SDK 2007.08 и Microsoft SDK 2003 SP 1. Движок старый и поэтому требует таких компонент как d3d.h, которые в современных DirectX SDK отсутствует.

Объекты предоставляемые движком основаны на общем классе под названием Widget. Каждый объект будь-то сцена или кнопка является классом Widget. Widget меняет свою функциональность благодаря добавлению к классу классов предков. От этого угроза коллизий не возникает, так как движок был грамотно написан.

В плане физики в отличии от Unity не имеет встроенного движка на подобие Box2D. Столкновение объектов вычисляется на уровне пересечении прямоугольников. Результатом этого стало, что многие, используя движок PopCap Framework, создают квесты и т.п., для которых не требуется большое кол-во вычислений физики.

Общая структура движка в плане логики и отрисовки похожа на предыдущие рассмотренные движки (см 1.5).

### 2.3.3 Требования игрового движка

Исходя из рассмотренных игровых движков можно прийти к выводу что простой игровой движок должен предоставлять такой функционал:

* Вывод графики
* Проигрывание Анимации
* Вычисление столкновений (коллизий) между несколькими объектами
* Основные объекты будущей игры с помощью, которой можно будет создать такие объекты как моб, бочка, и т.д.
* Движок навигации и переходов между игровыми сценами.
* Воспроизведение звука и музыки
* Сценарии, скрипты

Для реализации вышеперечисленных функционалов требуются использовать графический движок, физический движок и скриптовый движок. Объекты предоставляемые движком должны уметь обрабатывать любые скрипты прикрепленные к ним.

## 2.3. Разработка игрового движка

При разработке игрового движка основной целью было создать движок, подходящий для интеграции с конструктором. В рассмотренных движках было ясно что основами движков является графика и физика. Первым шагом к достижению целей, поставленных в предыдущей подразделе, является выбор или разработка графических и физических движков.

### 2.3.1 Разработка компоненты графики

Для графического движка были поставлены следующие требования.

1. Возможность выводить графику. Проще говоря, должен иметь метод draw(image, posX, posY), который рисует image по координатам posX и posY. Также из примера вытекает, что движок должен иметь такие классы как Image, т.е. те компоненты необходимые для вывода графики.
2. Проигрывание анимации. В большинстве предлагаемых графических движков отсутствует компонент, предназначенный для проигрывания анимаций. Дело в том, что анимация к графике не относится, а является эффектом.

Для отрисовки графики была использована библиотека GameEngine2D,, которая предоставляет функционал для создания простой двумерной игры. Она написана на языке C# и входит в состав PSM SDK. Некоторые ее аспекты основаны на библиотеке Cocos2D, свободном 2D-движке, использующем OpenGL для аппаратного ускорения графики [15]. Она использует похожие ключевые объекты, такие как Director, Sheduler и ActionManager. Библиотека GameEngine2D доступна вместе с исходным кодом.

**Что делает GameEngine2D**

GameEngine2D делится на два пространства имен:

Sce.Playstation.HighLevel.GameEngine2D.Base содержит различные функции, такие как математические библиотеки (одна из которых конфликтует с системной библиотекой Math), функции рендеринга (зарисовки), вспомогательные функции обработки ввода, камера, шрифтовые сопоставления, набор функций для работы с матрицами и т.д.

Sce.Playstation.HighLevel.GameEngine2D содержит “Игровой движок” (граф сцены, диспетчер, действия, и т. д.) который полностью собран из  GameEngine2D.Base.

GameEngine2D не предоставляет функционал обнаружения коллизий, за исключением простых правил из GameEngine2D.Base.Math, а также функционал физики, звука, сети и т. д. Также нет объектов для элементов пользовательского интерфейса, хотя некоторые  UI объекты можно сделать, используя простейший вывод текста и спрайт.

**Базовое использование GameEngine2D**

После подключения библиотеки GameEngine2D надо инициализировать Director. Director это одноэлементное множество, которое позволяет получить доступ к глобальным элементам Director.Instance(). Он должнен быть инициализирован Director.initialize().

**Создание пустой сцены.** После инициализации создается сама сцена. В простейшем случае мы можем использовать только одну сцену, SceneObject, и соответственно один узел. Сцена (Обычно говорят графическая сцена) представляет собой этакую театральную сцену, актерами которого выступают объекты предоставляемые библиотекой. Так же сцена имеет свою систему координат.

**Цикл**(см. раздел 1.5). Можно так же сделать собственный Update, вызывая поочередно методы Update(), Render(), и PostSwap() объекта класса Director.

**О структуре сцен(Scenegraph)**

**Общая структура.** Структура сцен(далее Scenegraph) используется для хранения иерархии вершин(узлов) игрового мира, и для обхода. Больше ни для чего не используется.

Корневой элемент каждого Scenegraph - это сцена. Навигация сцены обрабатываются стеком сцен через Director.Instance().

Узел имеет тип Node и хранит список дочерних узлов(Children), а также не более одного родительского узла(parent). Children и Parent можно изменять с помощью методов AddChild(), RemoveChild(), RemoveAllChild().

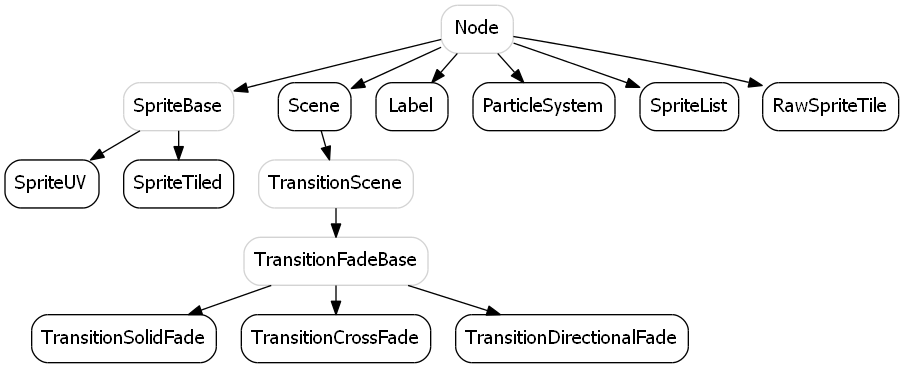
**Update вершин**

Каждый узел в Scenegraph может иметь “Действия(Actions)”. Управляет действиями ActionManager.Instance, но в основном управляются из членов узла. В качестве действия может быть перемещение объекта.

**Преобразование вершин(Node)**. Корневой объект каждой вершины графа сцены имеет свойства для геометрического преобразования, такие как положение, вращение, масштабирование.

**Иерархия вершин**

На рис. 1 приведена стандартная иерархия вершин Scenegraph.



**Рис. 1**. Иерархия вершин Scenegraph

### 2.3.2 Разработка класса AnimationSprite

Для создания двумерной игры анимацию приходится программировать много раз. Данный класс разработан для облегчения этой работы.

Спрайт (англ. *Sprite* — фея; эльф) — графический объект в компьютерной графике, чаще всего — растровое изображение, свободно перемещающееся по экрану. То есть легче всего воспринимать спрайт как перемещающуюся в пространстве проекцию какого-то объёмного тела так, что разница незаметна. Для анимации спрайтов используются поочередно отображаемые кадры, соединенные в одно большое изображение. Размер спрайта должен быть кратен степени двойки(Например 128х128, 256х256 и  т.д.)

Класс основан на библиотеке GameEngine2D т. к. она предоставляет множество функций визуализации. От GameEngine2D используется класс Sprite для удобности вывода картинки.

**Интерфейс класса:**

1. AddAnimation(string animationName, TextureInfo textureInfo, int indexCount, double Delay)

Данная функция добавляет анимацию и хранит его в поле Animations.

Входные данные:

* animationName - Имя анимации
* textureInfo - данные о текстуре
* indexCount - кол-во кадров.
* Delay - Время задержки каждого кадра.

2. PlayAnimation(String Name, PlayType playType);

Функция для воспроизведения той или иной анимации.

Входные данные:

* Name - Имя воспроизводимой анимации
* playType - Тип анимации (см. ниже)

3. Update()

В данной функции происходит логика смены кадра.

Входные данные:

* Отсутствуют

**Проблемы:**

Во время разработки были встречены проблемы, которые перечислены ниже:

1. Размеры спрайта не должны превышать 2048х2048

2. Отсутствие глобального таймера в стартовом проекте. В API XNA на основных функциях имеется параметр GameTime который помогает вычислять время от начала функции, от старта компоненты и от старты игры, а также реальное время что очень помогает при работе с временем.

3. Нумерация спрайтов в классе SpriteTile нумерация идет снизу-вверх слева-направо

**Решение проблем:**

1. Разделить спрайты определенных действий(например анимации бега в отдельную текстуру, стрельбы в другую)

2. В классе GameEngine2D имеется свой простой таймер Timer. Используя ее сделал аналог из GameTime из API XNA.

3. Пришлось переделывать спрайты.

**Для класса AnimationSprite разработаны следующие вспомога­тель­ные объекты:**

1. Структура под названием Animation.

Структура состоит из текстуры TextureInfo, в котором храниться картинка спрайт определенной анимации, переменная int (indexCount), где хранятся количество кадров (фреймов), и  вещественного числа (double), которое есть время (Delay) в секундах т.е. время перехода между кадрами.

В общем, Animation представляет собой свойства какой-нибудь конкретной анимации.

Например, если у персонажа будет анимация бег, то в TextureInfo будет храниться картинка с кадрами анимации бега. В index Count кол-во кадров, из которых состоит анимация бега, и Delay, время переключения между кадрами. Для каждой анимации (прыжка, удара и т.д.) будет создан отдельный экземпляр структуры Animation.

2. Перечисляемый тип PlayType. Тип проигрываемой анимации.

Состоит из Cycle и Once. Cycle говорит о том что анимация проигрывается циклично (дождь, огонь и т. д). Once для разовой анимации (Удар, прыжок и т. д).

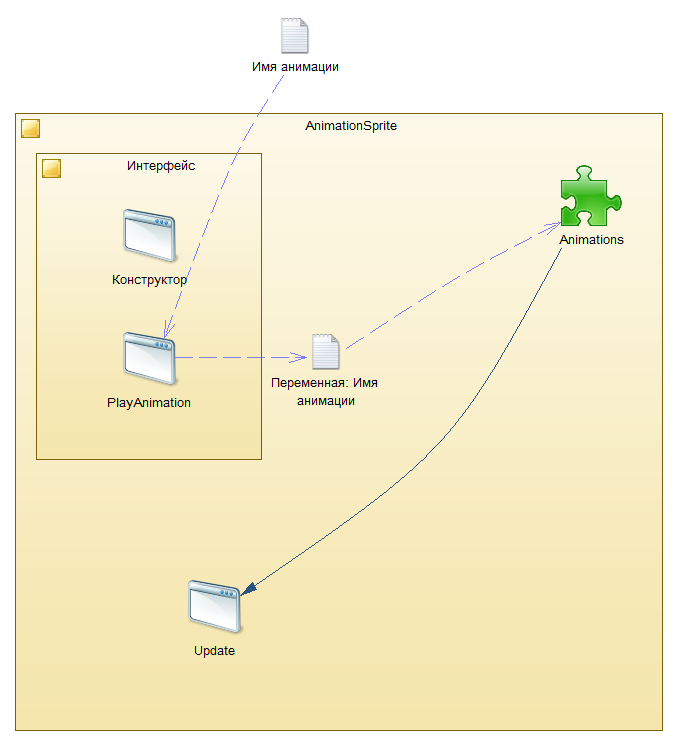


Рис.1. Внутренняя структура класса AnimationSprite

**Как работает класс AnimationSprite**

Через конструктор, к классу передается объект SpriteTile. SpriteTile это объект в котором храниться текстура, позиция, размеры, и даже текущая ячейка. Именно с объектом SpriteTile будет работать наш класс AnimationSprite.

**Структура класса**

Анимации хранятся в “словаре” Dictionary<string, Animation>(Сперва имя анимации и потом структура Animation) под названием Animations.

Для проигрывания анимации вызывается метод PlayAnimation, а обработка перехода между кадрами происходит в методе Update.

В методе PlayAnimation указываем имя и тип анимации. Потом PlayAnimation берет и передает данные в метод Update.

Update воспроизводит анимацию так:

Из “словаря” Animations по имени находится анимация и каждые Delay времени увеличивается счетчик currentColumn(начинается от нуля). По счетчику находится индекс ячейки(SpriteIndexes) и потом передается в Sprite.TileIndex1.

Метод Update должен вызываться в методе Update(или в его аналоге) в основном классе(AppMain.cs) проекта.

Таким образом, получается спрайтовая анимация.

### 2.3.3 Разработка компоненты физики

Следующим элементом костяка игрового движка является физика. Без физики наши анимированные объекты не смогут двигаться и взаимодействовать между собой.

**Физический движок** — компьютерная программа, которая производит компьютерное моделирование физических законов реального мира в виртуальном мире, с той или иной степенью аппроксимации. Чаще всего физические движки используются не как отдельные самостоятельные программные продукты, а как составные компоненты (подпрограммы) других программ.

Основным критерием к физическому движку для игры жанра платформера, является возможность вычисление динамики твердых тел в двумерном пространстве.

К этому критерию соответствует такой движок Physics2D, входящий в состав в PSMSDK.Некоторые ее аспекты основаны на библиотеке Box2D, написанной на [языке программирования](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)[C++](http://ru.wikipedia.org/wiki/C%2B%2B) и распространяющейся на условиях [лицензии zlib](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%86%D0%B5%D0%BD%D0%B7%D0%B8%D1%8F_zlib). Библиотека Physics2D доступна вместе с исходным кодом.

**Что делает Physics2D**

Sce.PlayStation.HighLevel.Physics2D предоставляет базовую модель 2D моделирования физики.

Physics2D не предоставляет функционал для рисования объектов. Так же как и GameEngine2D имеет собственную сцену со своей системой координат. В отличие отличии от сцены GameEngine2D объектами сцены выступают не графические картинки, а физические тела.

**Процесс создания пустой сцены симуляции физики**

Процесс создания пустой сцены моделирования физики.

1. Создать сцену класса наследуемую от PhysicsScene
2. Создаем пустую сцену моделирования Override InitScene ()
3. Вызываем InitScene() внутри конструктора сцену класса

**Минимальные настройки для сцены моделирования физики готовы.**

Но нет никаких физических объектов внутри сцены, и моделирование ничего делать не будет, вообще.

Теперь мы должны положить необходимые твердые тела в качестве начального состояния внутрь сцены InitScene ().

**Начальная настройка твердых тел**

Каждая тело представляется в виде класса, которая хранятся в массивах. Основные процессы начальной настройки твердого тела.

1. Создать формы
2. Создание твердого тела, связанного с формой
3. Создание формы столкновения

В основном форма должна быть выпуклой, включая выпуклые сферы, коробки и многоугольники.

Максимальное число вершин выпуклой формы 30. Но при изменении исходного кода, мы можем изменить верхнее ограничение.

Нельзя создать вогнутую форму. Вогнутая форма должна рассматриваться как соединение нескольких твердых тел.

Формы регистрируются в sceneShapes.

В то же время, не надо забывать устанавливать numShape равное количеству созданных форм.

**Пример создания коробки**

Vector2 wall\_width = new Vector2 (50, 8);

sceneShapes [0] = new PhysicsShape (wall\_width);

**Пример создания сферы**

floatsphere\_width = 2.0f;

sceneShapes [3] = new PhysicsShape (sphere\_width);

**Пример создания выпуклой формы**

Создаем 10 случайных точек которые будут вершинами новой формы.

Vector2[] test\_point = new Vector2[10];

for (inti = 0; i< 10; i++)

{  
       test\_point[i] = new Vector2(rand\_gen.Next(-1000, 1000), rand\_gen.Next(-1000,

1000)) \* 2.0f / 1000.0f;

}  
sceneShapes[4] = PhysicsShape.CreateConvexHull(test\_point, 10);

CreateConvexHull создает выпуклую оболочку из нескольких случайных.

**Количество форм**

Мы можем увеличивать количество форм по одному, но тут установили numShape вручную.

*numShape = 5*

Но это только оформление, регистрация форм.

Затем мы создадим твердые тела на основе зарегистрированных форм столкновения.

**Создание твердого тела через объявленные формы**

Есть несколько типов твердых тел.

**Динамическое твердое тело**

* Обычное твердое тело
* Имеет скорость и ускорение и может двигаться на сцене

**Статическое твердое тело**

* Вообще не двигается от начала сцены и до его конца
* Рассчитано исходя из предположения, что она имеет бесконечную массу

**Кинематическое твердое тело**

* Устанавливается в состояние покоя или может быть временно перемещено
* Рассчитано исходя из предположения, что она имеет временную бесконечную массу.

**Импульс твердого тела**

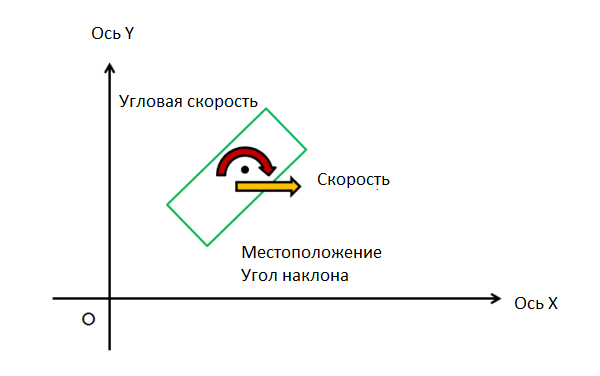
* Используется с целью определения наличия любых твердых тел, войдя в некоторую область в сцене
* Обнаруживает столкновение, но не реагирует на столкновение

Прежде всего, необходимо выяснить, какой тип твердого тела мы хотели бы создать.

Статические твердого тела используется при создании стены или пола, которые не двигаются вообще. С другой стороны, динамическое твердое теле используется при создании шара, который катится по полу.

**Атрибут, используемый для начальной настройки твердого тела**

* Масса
* Местоположение
* Угол наклона
* Скорость
* Угловая скорость



// Создание статической стены

{

// newPhysicsBody( "форматела", "масса тела(кг)" )

sceneBodies[numBody] = new PhysicsBody(sceneShapes[0], PhysicsUtility.FltMax);

// Установка позициии угла наклона

sceneBodies[numBody].position = new Vector2(0, -wall\_height.Y);

sceneBodies[numBody].rotation = 0;

// Индекс формы

sceneBodies[numBody].shapeIndex = 0;

numBody++;

sceneBodies[numBody] = new PhysicsBody(sceneShapes[1], PhysicsUtility.FltMax);

sceneBodies[numBody].position = new Vector2(wall\_width.X, 0);

sceneBodies[numBody].rotation = 0;

sceneBodies[numBody].shapeIndex = 1;

numBody++;

sceneBodies[numBody] = new PhysicsBody(sceneShapes[1], PhysicsUtility.FltMax);

sceneBodies[numBody].position = new Vector2(-wall\_width.X, 0);

sceneBodies[numBody].rotation = 0;

sceneBodies[numBody].shapeIndex = 1;

numBody++;

sceneBodies[numBody] = new PhysicsBody(sceneShapes[0], PhysicsUtility.FltMax);

sceneBodies[numBody].position = new Vector2(0, wall\_height.Y);

sceneBodies[numBody].rotation = 0;

sceneBodies[numBody].shapeIndex = 0;

numBody++;

}

### 2.3.4 Разработка класса Game Physics Engine для физики

Как видно из предыдущего подраздела, для создания даже простейшего объекта приходится писать очень много. А так как при написании игры надо будет вызывать очень много объектов и объектов не самых простых, нужен такой класс, который упростит эту работу. Был разработан класс GamePhysicsEngine, который облегчил создания физических объектов через одну вызываемую функцию, передав свойства объекта через параметры.

**Интерфейс класса:**

1. addbox(float width, float height, float X, float Y);

Функция добавляет статическую коробку.

Входные данные:

* width - ширина
* height - высота
* X - местоположение п ооси X
* Y - местоположение по оси Y

2. addbox(float width, float height, float X, float Y, float mass);

Функция добавляет динамическую коробку.

Входные данные:

* width - ширина
* height - высота
* X - местоположение по оси X
* Y - местоположение по оси Y
* mass – масса

3. addcircle(float r, float X, float Y);

Функция добавляет статическую сферу.

Входные данные:

* r - радиус
* X - местоположение по оси X
* Y - местоположение по оси Y

4. addcircle(float r, float X, float Y, float mass);

Функция добавляет динамическую сферу.

Входные данные:

* r - радиус
* X - местоположениепооси X
* Y - местоположение по оси Y
* mass – масса

5. addconnect(uint ind1, uint ind2, float X, float Y, Vector2 axis1, Vector2 axis2, float axisXL, float axisXU, float axisYL, float axisYU, int angle);

Функция которая соединяет между собой 2 тела.

Входные данные:

* ind1 - индекс 1го объекта
* ind2 - индекс 2го объекта
* X - местоположение по оси X
* Y - местоположение по оси Y
* axis1 - возможность перемещения по оси X
* axis2 - возможность перемещения по оси Y
* axisXL - нижний предел по оси X
* axisXU - верхний предел по оси X
* axisYL - нижний предел по оси Y
* axisYU - верхний предел по оси Y
* angle - разрешенный угол поворота

6.addline(Vector2[] arraypoint, float X, float Y);

Функция добавляет форму созданную из линий

Входные данные:

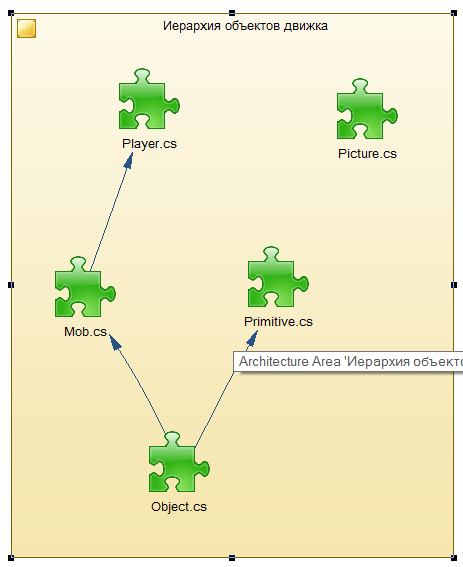
* arraypoint - массив состоящий из координат точек
* X - начальное местоположение по оси X

Y - начальное местоположение по оси Y

### 2.3.5 Разработка стандартных объектов движка

Так как в игре есть очень много разных объектов, то соответственно движок должен предоставлять возможность их создания. Для этого были придуманы такие объекты как:

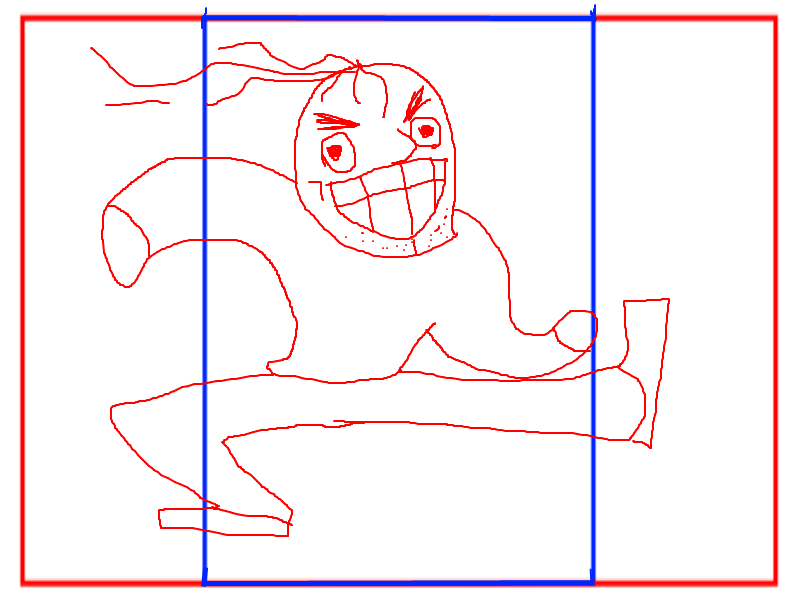
1. Object.cs
2. Mob.cs
3. Primitives.cs
4. Picture.cs



**Рис. 1.** Иерархии Объектов Движка

Класс Object.cs находиться на низшем уровне иерархии классов. Представляет поля и методы для объединения графического и физических компонентов. Так как физический движок не может выводить графику, поэтому используется возможности графического движка. Создается физический объект и поверх него рисуется графика (см. рис. 2). Для этого мы должны к координатам физики соотнести координаты графики.

Абстрактно это можно представить как наложением двух слоев сцен. Действительно же это является двумя разными сценами, где в физической сцене вычисляется координаты, а в графической же рисуется. Прежде чем рисовать надо сделать дискретизацию координат учитывая особенности физической сцены. Следовательно, класс должен иметь такие поля как указатель на физический компонент GamePhysicsEngine и на графический объект Scene.



**Рис. 2.** Пример наложения графического объекта к физической.

На рисунке представлен пример как должен выглядеть наложение. Синий прямоугольник это объект физической сцены. Красная и нарисованная на нем объект являются объектами графической сцены.

**Object.cs**

Класс Object.cs будет являться основой для последующих классов. Следовательно, этот класс должен хранить все основные поля и методы нужные классам наследникам. Так как в этом классе не происходит создание самого физического объекта, то его реализация не будет описана на этом уровне.

**Поля класса.** Так как поля будут доступны как privateв наследуемых классах, то модификаторы доступа будут protected.

1. protected AnimationSprite Animations

Объект AnimationSprite.cs предоставляющий функционал проигрывания анимаций.

1. protected int collisionObjectId;

Уникальный номер другого физического объекта, с которым сталкивается объект.

1. private double elapsedTime = 0;

Время в миллисекундах одного тика программы.

1. protected GamePhysicsEngine gamePhysicsEngine;

Указатель на физическую сцену

1. protected bool left = false;

Булевая переменная нужная для определения стороны взгляда объекта.

1. protected int physicsId;

Уникальный номер объекта в массиве физических объектов.

1. protected Vector2 physicsSize;

Размер объекта в координатах физической сцены.

1. protected Vector2 positionPhysics;

Позиция в физической сцене.

1. protected Vector2 positionSprite;

Позиция в графической сцене.

1. protected float scaleRatio = 30;

Является ключевой переменной для решения "наложения" физических и графических сцен.

1. protected Scene scene;

Указатель на графическую сцену.

1. protected SpriteTile sprite;

Объект, предоставляемый движком GameEngine2D нужный для класса анимаций AnimationSprite.

1. protected Vector2 spriteSize;

Размер объекта в координатах графической сцены.

**Методы,** предоставляемые объектом.

1. AddAnimation(string name, TextureInfo \_textureInfo, int frameCount, float Delay)

Метод, который добавляет анимации в animations.

1. public Vector2 ConvertSpritePosToPhysicsPos(Vector2 spritePos)

Преобразование координат графической сцены в координаты физической сцены.

1. public Vector4 GetCollision()

Возвращает четырехмерный вектор X1, X2, X3, X4. Где X1 равный 1 - есть коллизия справа. X2 - слева, X3 - сверху и X4 - снизу.

1. public bool GetLeft()

Возвращает значение переменной left

1. public void GoJump(float speed)

Вызвав эту функцию объект должен подпрыгнуть вверх по законам физики предоставляемый движком GamePhysicsEngine.

1. public void GoLeft(float speed), public void GoRight(float speed)

Движение объекта направо и налево.

1. public virtual int IdBodySynch()

Синхронизирует старый physicsId c новым. Объекты(например Монстры, предметы и т.д.) могут быть уничтожены во время игры, поэтому надо своевременно выгружать их из памяти. Объекты хранятся в списках, и после удаление нумерация объектов будет изменена. Поэтому надо все время обновлять PhysicsId. Если их не обновлять, то при вычислений коллизий с объектом с номерами id1 и id2 возникнет ошибка.

1. public void Initialize()

Здесь в дальнейшем будут инициализированы скрипты.

1. public Object ()

Конструктор объекта.

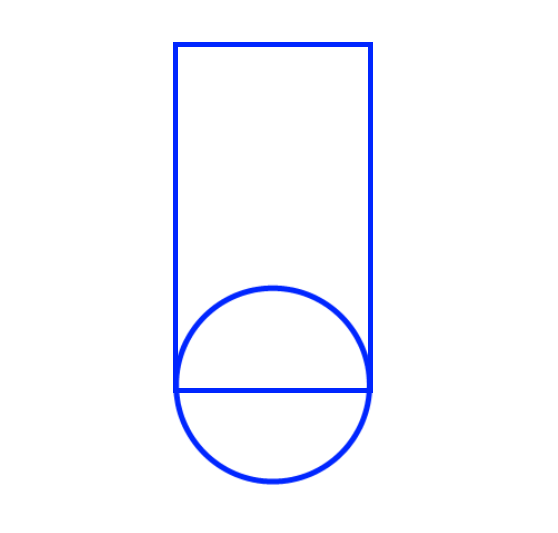
1. public int PlayAnimation(string name, int indexStart, int indexEnd, PlayType playType)

Проигрывание анимаций. Функционала предоставляемый из класса AnimationSprite.

**Mob.cs**

Исходя из названия, класс мобов. Здесь уже должен быть описан процесс создания физического объекта. Что он будет из себя представлять(его форма).

Так как выбранный физический движок учитывает такие вещи как трение, упругость и т.д. то форма представляет собой объединение двух физических объектов круг и прямоугольник.



**Рис. 1.** Форма физического объекта

Выбор именно такой формы объясняется тем, что касаться прямой или наклонной земли будет только в одной точке. Этот выбор хорош тем, что факторы трения не будут сильно препятствовать во время игры.

**Поля класса:**

1. PhysicsBody actsph\_body;

Из-за того что наш объект представляет собой соединение двух объектов(круг и прямоугольника) эти объекты будут созданы отдельно. Это т объект представляет собой тело круга.

1. PhysicBody actor\_body;

Тело прямоугольника.

1. uint actsph\_body\_index;

Индекс круга. ЭтакийphysicsIdэтого объекта

1. uintactor\_body\_index;

Индекс прямоугольника

Далее идут переменные для хранение стандартных свойств моба.

1. protected Vector2 boxPhysicsSize;

Размер прямоугольника

1. protected float circlePhysicsSize;

Размер(Диаметр) тарелки

1. private Vector2 boxSpriteSize;
2. private float cirlceSpriteSize;

Размеры физических объектов моба по координатам графической сцены

1. protected int speed;

Скорость

**Методы предоставляемые классом**

1. public bool CheckCollisonObj1WithObj2(Vector2 pos1, Vector2 pos2, int deltaX, int deltaY)

Вычисление коллизий между объектами. Если между объектами возникнет коллизия то возвращаемая значение будет истинной.

1. public void CreateMob(Scene \_scene, GamePhysicsEngine \_gamePhysicsEngine, Vector2 \_boxSpriteSize, float \_circleSpriteSize, Vector2 \_spriteSize, Vector2 \_position, float mass, int \_collisionFilter)

Самый важный метод этого движка. Метод создающий физический объект моба и привязка к графической сцене.

1. public override int IdBodySynch()

Метод синхронизирующий physicsId и Idв массиве объектов физической сцены. Это функция будет вызываться при каждом тике логики.

1. public Mob ()

Конструктор класса

1. public void MobUpdate()

Метод вызываемый в методе Updateклассах наследниках. В этом методе описывается дискретизация координат графического и физического сцен.

Далее следующим классом является класс Player.Не смотря на название, каждый моб это экземпляр класса Player. В этом классе много полей свойств важных именно для игрока. Например, отключение контроля над персонажем, неуязвимость и т.д. Также метод описывающий алгоритм прикрепление камеры к игроку, т.е. контролирующий чтобы игрок не ушел за пределы сцены.

**Поля класса**

1. private Camera2D camera;

Объект камеры игровой сцены

1. private Vector2 camPos;

Позиция камеры в координатах графической сцены

1. private bool isControlled;

Переменная определяющий контроль над игроком

1. public GamePadData kbState;

Объект, где храниться состояния геймпада. Этот объект определяет нажат ли кнопка и как именно она была нажата (Нажата, Отпущена, Нажат и т.д.)

**Методы класса:**

1. public bool DownButton(GamePadButtons button)

Метод, возвращающий истину на нажатие кнопки

1. public Player (GameScreen \_level, string levelName, Scene \_scene, GamePhysicsEngine \_gamePhysicsEngine, Vector2 \_boxCollideSpriteSize, float \_circleCollideSpriteSize, Vector2 \_spriteSize, Vector2 \_position, float mass, int \_collisionFilter, int \_maxHealth, string scriptFileName)

Конструктор класса Player. Помимо параметров которые нужны для CreateMob класс предка, здесь еще нужны параметры жизни(health) и полный путь к файлу скрипта.

1. public bool PressButton(GamePadButtons button)

Метод возвращающий истину если кнопка нажат.

1. public void Update()

Основной метод логики. В этом методе будет описана логика моба т.е. реакция на нажатие кнопки, реакция если закончился health и т.д.

1. public void UpdateCam(intdeltaX, intdeltaY)

Здесь описывается логика прикрепление камеры к игроку.

Следующий класс это Primitive. нужный для создания примитивных объектов. Этот класс следует использовать для создание игровых объектов состоящих из одного физического объекта. Например, таким объектов может быть мяч, ящики и т.п.

Особенностью этого класса является отсутствием полей, потому что функционал класса не такой большой, и наличием двух конструкторов. Первый конструктор предназначен для круглых объектов, а вторая для прямоугольных.

**Методы класса:**

1. public override int IdBodySynch()

Функционал этого метода такой же, как в Mob. Синхронизация PhysicsId с номером в массиве объектов физической сцены.

1. public Primitive (Scene \_scene, GamePhysicsEngine \_gamePhysicsEngine, float radius, Vector2 position, float mass)

Конструктор для круглых объектов.

1. public Primitive (Scene \_scene, GamePhysicsEngine \_gamePhysicsEngine, Vector2 \_size, Vector2 position, float rotation, float mass)

Конструктор для прямоугольных объектов. Различия от первого конструктора в третьем параметре. Первом указывается лишь радиус, а во втором вектор, в котором указан высота и ширина объекта.

1. public void Update()

Метод логики объекта. В отличие от MobUpdate здесь еще и синхронизируются углы наклона объектов.

Следующий класс это Picture. Класс, состоящий только из графического объекта. Такой класс обычно нужен для эффектов или картинок фона.

**Поля класса:**

1. private SpriteTile sprite;

Объект графического движка, отвечающий за вывод графики.

**Методы класса:**

1. public Picture (Scene \_scene, TextureInfo \_textureInfo, Vector2 \_size, Vector2 \_position)

Конструктор класса.

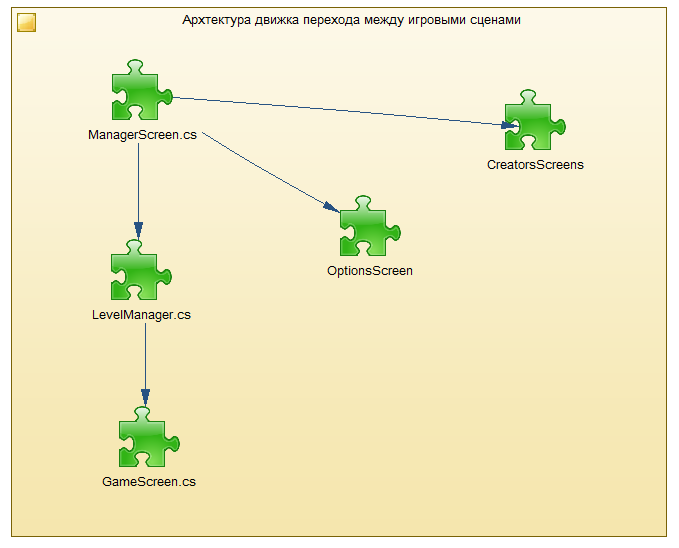
Выше перечислены и разобраны основные объекты, которые может создать и обрабатывать наш игровой движок.

### 2.3.6 Разработка движка навигации между игровыми сценами

Одним из важных объектов игрового движка является объект игровой сцены. Игра состоит из нескольких уровней, которые являются игровыми сценами. Следовательно, нужна реализация алгоритма для перехода между игровыми сценами. Для реализации перехода были разработаны два класса так называемых менеджеров:

* Первый класс менеджерManagerScreen.cs нужен для навигации между основными сценами(Под основными сценами подразумевается такие вещи как "Меню", "Опции", "Авторы" и т.п.).
* Второй класс менеджер Level Manager.csбудет нужен для реализации перехода между уровнями игры.

А саму игровую сцену будет представлять класс Game Screen.



**Рис. 1.** Архитектура движка перехода между сценами

ManagerScreen сначала запустит LevelManager. Потом LevelManager.cs запустить первый уровень игры.

Из разработанной архитектуры конструктора следует, что игровой движок должен прочитать проект - файл. Из проект файла движок узнает какие объекты должны быть реализованы. А как движок узнает, что существует проект файл? Имя проект файла храниться в GameInit.script. Значит движок должен прочесть первым этот файл. LevelManager прочтет этот файл и создаст уровни игры.

**LevelManager**

Первый класс который был разработан это LevelManager. Класс, который на основе прочитанного создающий, контролирующий их завершение и запускающий новых уровней. Этакий менеджер уровней игры. Параметры каждого уровня будут хранится в списке из массивов String(Dictionary<string, sting[]>). Это обусловлено тем, чтобы не забивать памятью полноценными классами GameScreen.

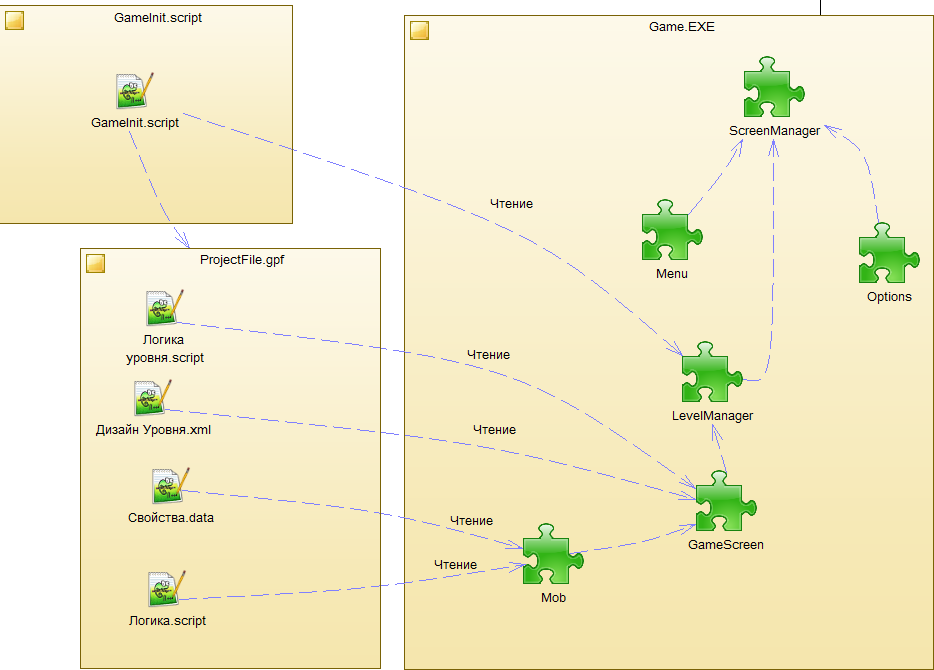


Рис 2. Принцип работы класса LevelManager.

Итого будет один класс который будет выгружаться и запускатьсяснова, но только с новыми параметрами.

**Поля класса:**

1. private string currentLevelName;

Имя уровня запущенный на данный моммент.

1. private bool isLevelManagerDispose;

Переменная указывающая на выгрузку класса

1. private bool isLevelNext;

Переменнаяуказывающая на запуск следующего уровня

1. private GameScreen level;

Сам объект уровня.

1. private Dictionary<string, string[]>levelParametersList;

Список типа Dictionary. Здесь храниться параметры уровней. Ключем словаря будет имя уровня. Например "LevelOne". Первый параметр это путь к XML файлу, где храниться координаты объектов игровой сцены, путь к файлу текстуры объекта и его свойства. Второй параметр это непосредственно путь к файлу скрипта уровня.

1. private Dictionary<string, string[]>mobParametersList;

Список типа Dictionary. Здесь храниться параметры мобов. Ключем словаря будет имя моба. Например "Elf". Первый параметр это путь к data файлу, где храниться свойства моба, путь к файлу текстуры спрайтов и его свойства. Второй параметр это непосредственно путь к файлу скрипта моба.

1. private string nextLevelName;

Имя следующего запускаемого уровня

**Методы класса:**

1. public void Initialize();

Здесь происходит процесс инициализации уровней игры. Сначала читает GameInit.script оттуда узнает имя проекта и начинает загонять параметры уровней в levelParametersList, а параметры мобов в mobParametersList. Параметры выше перечисленных объектов записаны в проект файле. После этого начинает запуск первого уровня исходя из GameInit.script.

1. public LevelManager (Director \_director, ManagerScreen \_managerScreen)// : base(director, managerScreen)

Конструктор класса.

1. public void NextLevel(string nextLevelName)

Метод выгружающий из памяти текущий уровень и запускающий следующий. Имя следующего файла записан в скрипте текущего уровня.

1. public void StartLevel(string levelName)

Метод вызываемый только один раз из GameInit.script.

1. public void Update()

Метод логики, где по циклу обрабатывается логика уровня.

**ScreenManager**

Следующий класс который был разроботан это ScreenManager. Этот класс обрабатывает переходы между основными игровыми сценами такими как "Новая игра", "Опции" и т.п. Стоит отметить что игровые сцены должны быть в виде классов из которых будет состоят ScreenManager.

**Поля класса**

1. private static Director director;

Объект из GameEngine2D

1. private bool isLevelManager ScreenDispose;

Переменная указывающая на выгрузку LevelManager

1. private LevelManager levelManager;

Сам LevelManager. По архитектуре он входит в состав ScreenManager.

1. private bool runLevelManager;

Переменная, указывающая на запуск работы LevelManager'а.

**Методы класса:**

1. private void Initialize()

Здесь происходит инициализация стартовой сцены игры.

1. publicManagerScreen (Director \_director)

Конструктор класса

1. public void screenDispose()

Метод, вызываемый из классов лежащих уровнем ниже, для выгрузки из памяти.

1. publicvoidUpdate()

Здесь запускаются классы наследники. НапримерLevelManager.

**GameScreen**

Последним разбором является класс GameScreen.cs. Класс представляющую саму игровую сцену. Как мы помним, при создании уровня, в LevelManager передавали параметры уровня. Этими параметрами были путь к файлу XMLи скрипта. Следовательно, этот класс должен уметь его прочесть и на его основе построить уровень.

**Поля класса:**

1. private Dictionary<string, Primitive> blocks;

Список типа Dictionary. В этом списке будут хранятся Primitiveобъекты прочитанные из XMLфайла требующие физику.Например платформы.

1. private GamePhysicsEngine gamePhysicsEngine;

Объект физики. Физическая сцена.

1. static bool isLevelEnd;

Переменная, указывающая на завершение уровня.

1. private string levelName;

Имя текущего уровня.

1. private Dictionary<string, string[]>mobParametersList;

Переданный список параметров от LevelManager. Здесь храниться параметры мобов.

1. static string nameNextLevel;

Имя следующего уровня.

1. private string nameScriptFile;

Путь к файлу скрипта для этого уровня. Здесь описана инициализация мобов т.е. расставление на карте, а также сценарий уровня.

1. private string nameXMLFile;

Путь к файлу XML. Здесь храниться координаты, путь к текстуре и свойствам объектов расставленных по карте.

1. private Dictionary<string, Picture>pictureList;

Прочитанные объекты, которые без физики, например фон, из XML файла будут описаны здесь.

1. private Dictionary<string, Player >playerList;

Собственно список мобов.

**Методы класса:**

1. public GameScreen (Director \_director, ManagerScreen \_managerScreen, string \_levelName, string \_nameScriptFile, string \_nameXMLFile, Dictionary<string, string[]> \_mobParametersList)

Конструктор класса

1. public void Initialize()

Здесь происходит создание графических и физических сцен для уровня, а также вызывается InitializeObjects.

1. public void InitializeObjects(string nameScriptFile)

В этом методе происходит заполнение списковDictionary<string, Picture>pictureListиDictionary<string, Primitive> blocks, а так же вызывается метод LoadXMLLevel.

1. public void LoadXMLLevel(string XMLFileName)

Исходя из названия загружаются объекты из XMLфайла.

1. public void MobInit(string Name, float posX, float posY, int health)

Метод вставки мобов. Вызывается из скрипта.

1. public void RunNextLevel(string nextLevelName)

Метод запускающий следующий уровень.

1. public void Update()

Метод, запускающий логику всех объектов уровней. Этот метод вызывается при каждом тике программы.

Все выше перечисленные объекты а также взаимодействие между собой есть движок навигации между игровыми сценами. Следующим важным фактором, который встречался до этого, является реализация поддержки скрипта в движок.

### 2.3.7 Интеграция скриптового языка Lua

**Сценарный, или скриптовый язык** — высокоуровневый язык программирования для написания сценариев (скриптов) — кратких описаний действий, выполняемых системой. Разница между программами и сценариями довольно размыта. Сценарий — это программа, имеющая дело с готовыми программными компонентами.

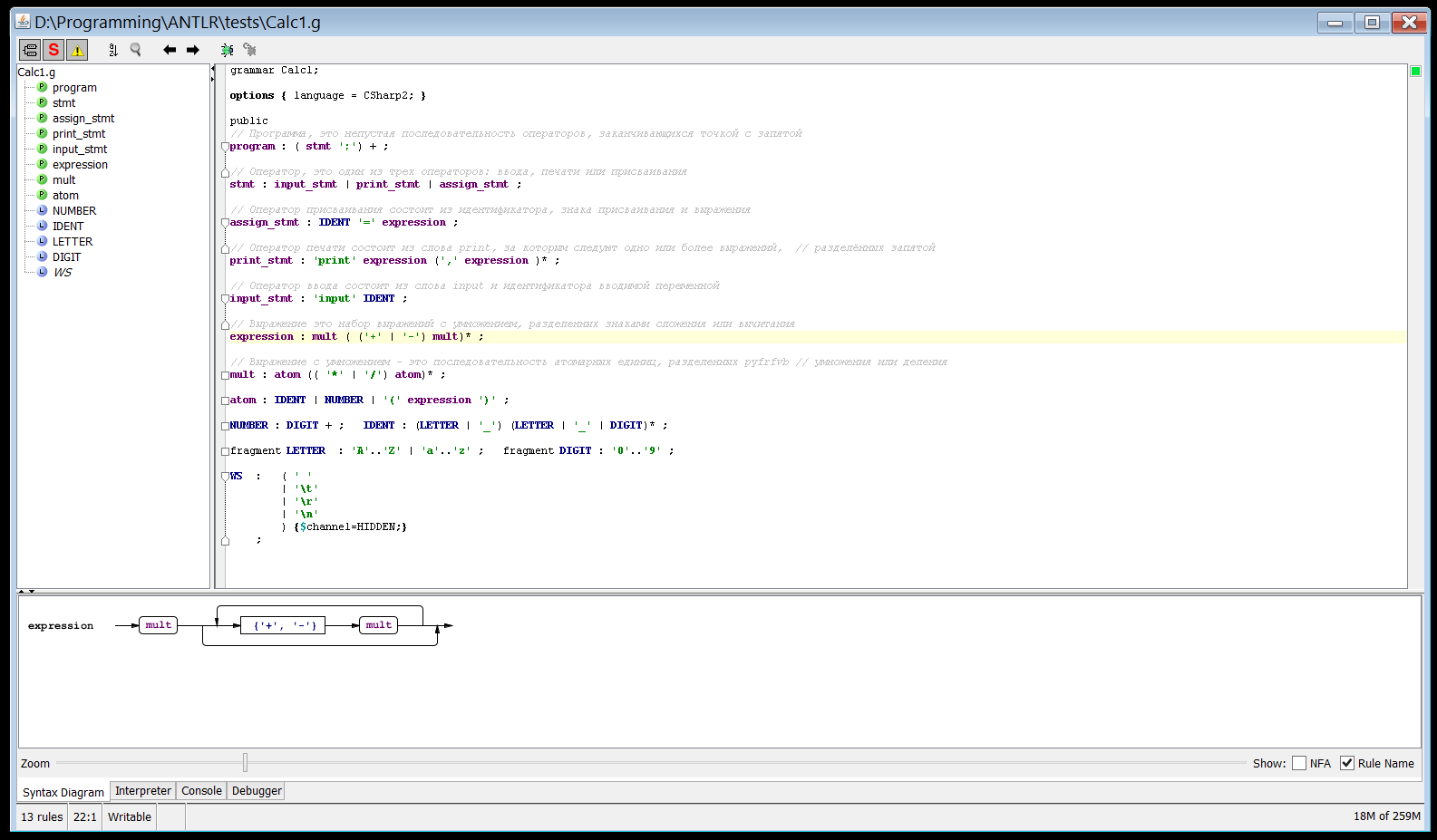
При создании игры скрипты используются во многом так же, как кинорежиссер использует сценарий — для контроля каждого аспекта поведения вашего «произведения». Игровые скрипты подобны программному коду, которые пишутся, во время создания игры, за исключением того, что скрипты являются внешними по отношению к игровому движку. Поскольку они внешние, можно быстро изменить скрипт не компилируя заново весь игровой движок.

Чтобы выбрать и изучить скриптовый движок, подходящий для наших целей, были рассмотрены следующее:

1. ANTLR
2. Stratchот LabView
3. Lua

**ANTLR**

**ANTLR** (***ANotherToolforLanguageRecognition*** — «ещё одно средство распознавания языков») — это не совсем движок, а генератор парсеров, позволяющий автоматически создавать программу-парсер (как и лексический анализатор) на одном из целевых языков программирования (C++, Java, C#, Python, Ruby) по описанию LL(\*)-грамматики на языке, близком к РБНФ(Расширенная форма Бэкуса - Наура). Позволяет конструировать компиляторы, интерпретаторы, трансляторы с различных формальных языков. Предоставляет удобные средства для восстановления после ошибок и сообщения о них. ANTLR — продолжение PCCTS (*PurdueCompilerConstructionToolSet*), который был разработан в 1989 г. Есть собственный IDE написанный на JavaANTLRWorks.



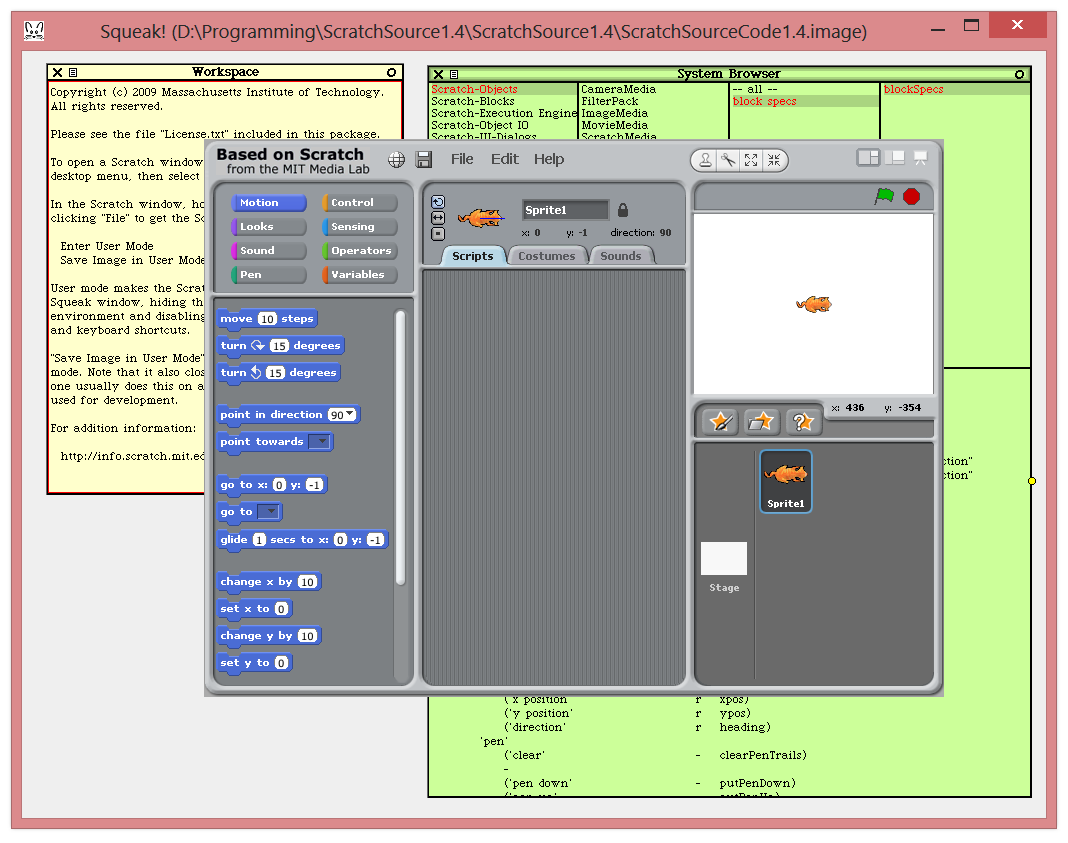
**Рис.1.** СкриншотANTLRWorks

На рисунке показана составление словаря и правила для программы калькулятор. В дереве в левой части написаны, с зеленой иконкой в вершинах, описаны правила. В синих поддерживаемый язык, т. е. словарь. После составления правил и словаря программа порождает два файла под поддерживаемый язык программирования. Файл лексер и парсер. Подключив их проекту можно прочитать язык под продуманную грамматику.

Хорош тем что можно составить любой язык под свою прихоть. Сложность заключается в том что грамматику придется сделать с нуля, что подразумевает очень много работы.

**Stratch от LabView**

Визуальная объектно-ориентированная среда программирования для обучения школьников младших и средних классов. Скретч создан как продолжение идей языка Лого и конструктора Лего. Написан на языке Squeak.



**Рис. 2.** Cкриншот Scratch

Основными компонентами Скретч-программы являются объекты-спрайты. Спрайт состоит из графического представления — набора кадров-костюмов и сценария-скрипта. Для редактирования костюмов спрайтов в скретч встроен графический редактор. Действие скретч-программы происходит на сцене размером 480×360 (условных) пикселей с центром координат в середине сцены.

Для программирования сценариев в скретче используется drag-and-drop-подход: блоки из Окна блоков перетаскиваются в область скриптов.

Предполагалось интегрировать в движок с целью облегчить написание скриптов, благодаря его графическому дизайну. Из-за того что он был написан на другом языке(smalltalk) предполагал очень много работы для понимания кода не говоря о расширении функционала.

**Lua**

Следующим, рассматриваемым скриптовым движком, является язык Lua.

Является интерпретируемым языком программирования, разработанный подразделением Tecgraf Католического университета Рио-де-Жанейро (*Computer Graphics Technology Group of Pontifical Catholic University of Rio de Janeiro in Brazil*). Разработанный интерпретатор является свободно распространяемым, с открытыми исходными текстами на языке Си.

Первыми в разработку компьютерных игр язык Lua внедрила компания LucasArts, начиная с игры GrimFandango. Авторы языка в своём докладе на конференции HOPL вспоминают, что в январе 1997 они получили сообщение от Брета Могилефски, главного разработчика GrimFandango, где он писал, что, прочитав о языке в статье 1996 года в Dr. Dobb’sJournal, он планирует заменить используемый ими самодельный скриптовый язык SCUMM на Lua. В результате им был создан игровой движок GrimE, используемый также более поздним квестом от LucasArts —EscapeFromMonkeyIsland.

В 2003 году в результате опроса на сайте GameDev.net Lua был признан самым популярным скриптовым языком для разработки игр.

Примером игры, программируемой с помощью Lua, является World of Warcraft. На языке Lua описываются уровни игры-головоломки Enigma. В играх S.T.A.L.K.E.R. используется Lua.

**Использование Lua**

Для использования в среде .NETиспользуется библиотека LuaInterface.dll. Библиотека отвечает за трансляцию кода Lua и конвертацию данных в рамках сред или языков программирования, поддерживающих общеязыковую исполняющую среду CLR. Она написана большей частью на C#, хотя в ней имеется и около 30 строчек на обычном С.

Подключается в проекте как отдельный компонент .Net.

Lua — язык с динамической типизацией (переменные получают типы в зависимости от присвоенных значений). Писать на нем можно как в императивном, так и в объектно-ориентированном или функциональном стиле. Вот простейшая программа выводящая слово "Hello world" на Lua:

--Моя первая программа на Lua

print('helloworld')

Двумя дефисами (--) обозначают комментарий. Текст пишется внутри одинарных кавычек.

**Синтаксис операторов языка**

Набор условных операторов и циклов довольно типичен и похож на язык Paskal.

Синтаксис условных операторов. Пример оператора if:

if a==0 then

print('a это ноль')

Для оператора типа switchизc# используется расширение оператора if. Пример:

if a==1 then

print(1)

elseif a==2

print (2)

elseif a==3

print(3)

Синтаксис циклов. Пример простого цикла:

for i = 1, 10 do

  print(i)

end

Пример цикла с предусловием

b = 5

while b> 0 do

  b = b - 1

end

Пример цикла с постусловием

repeat

  b = b + 1

until b >= 5

В выражениях также используются операторы над переменными:

* присваивание: x = 0
* арифметические: +, -, \*, /, % (остаток от деления), ^ (возведение в степень)
* логические: and, or, not
* сравнение: >, <, ==, <=, >=, ~=  
  конкатенация строк (оператор «..»), напр.: s1=»hello»; s2=»world»; s3=s1..s2  
  длина/размер (оператор #): s=»hello»; a = #s (‘a’ будет равно 5).  
  получение элемента по индексу, напр.: s[2]

**Типы данных**

Стандартные типы данных предоставляемые языком.

* nil альтернатива null
* Булевы числа
* Числа (numbers) — не делятся целые/вещественные.
* Строки — похожи на строки в паскале
* Функции — функции можно передавать ввиде переменных
* Поток (thread)
* Произвольные данные (userdata)
* Таблица (table)

Тип userdata служит для хранения произвольных данных (структур Си) в переменных Lua.

**Таблицы**

Таблицы это динамические ассоциативные массивы, на которых фактически основана вся синтаксическая мощь языка:

tbl = {}

print(type(tbl))

tbl[0] = {} -- вложенная таблица

tbl["str"] = 2

tbl[1.23] = nil -- это удаление элемента (если он есть), а не создание

fori,j in pairs(tbl) do -- обход таблицы

print(i,j)

end

print(tbl[2]) -- нет такого элемента, это nil

tbl2 = tbl -- копирования не происходит!

print(tbl2, tbl) -- выдаст один и тот же адрес, это один объект.

**Функции**

Тип **function** соответствует функциям. Функции в Lua могут быть записаны в переменные, переданы как параметры в другие функции и возвращены как результат выполнения функций.

Lua позволяет перегружать операции для таблиц и userdata с помощью **метатаблиц**. Метатаблица представляет из себя обычную таблицу, в качестве ключей которой используются строки (события), а в качестве значений — методы (обработчики). C метатаблицей может быть связана еще одна метатаблица (ключ "\_\_metatable"), и, если событие не определено в первой метатаблице, оно ищется во второй, и т.д. Все глобальные переменные являются полями в специальных таблицах Lua, называемых таблицами окружения (environment tables).

После рассмотренных скриптовых движков было решено использовать Lua по ряду причин:

* это зрелый, стабильный и популярный язык; Lua используется в промышленных приложениях, и в настоящее время это ведущий язык сценариев в играх;
* это один из самых производительных скриптовых языков в принципе;
* Lua существует для Linux, Windows и MacOS, а также может быть запущен и на мобильных устройствах;
* Lua — компактный и легко встраиваемый язык; имеется документированный API, который позволяет осуществить сильную интеграцию с кодом, написанным на других языках; Lua использовался, чтобы расширить программы, написанные на C, C++, Java, C#, Smalltalk, Фортране, Ада, и даже Perl и Ruby; точно так же можно расширить Lua библиотеками, написанными на других языках;
* хотя Lua — не объектно-ориентированный язык в строгом понимании этого термина, он обеспечивает специальные мета-механизмы для того, чтобы реализовать классы и наследование; эти мета-механизмы экономят сущности и сохраняют язык маленьким, позволяя при этом расширить его семантику;
* поставка языка компактна;
* есть возможность вызывать функции, описанные в коде исполняемого файла, из скрипта.

**Интеграция Lua скрипта в движок**

В процессе интеграции возникали проблемы с поиском поддерживаемой библиотеки под платформы PSM. Сообществом портала разработчиков Sony(psm.playstation.com) разрабатывался проект KopiLua, которая является портированием LuaInterface.dll. Позднее Sony добавила проект к официальным библиотекам комплекта PSMSDK.

Пример инициализации в коде движка.

lua = new Lua();

lua.DoFile(scriptFileName);

Вызов в цикле(см. 1.5)

lua.DoString("Main()");

В движке скрипт Lua будет использоваться для описания логики объектов и их инициализации. Было решено сделать два разных архитектуры скриптов. Для скрипта уровней и мобов.

Архитектура скрипта уровня было решено разделить на две основные функции:

* Initialize()

В этой функции происходит расстановка объектов.

* Update()

В этой части скрипта обрабатываться сценарий (логика)уровня.

Для скрипта объектов этого не требуется. Объясняется тем, что процесс инициализации будет описана в data файле. Есть только одна функцияMain(), в котором обрабатывается логика.

В ходе разработки логики объектов были поставлены следующие функции, вызываемые из скрипта:

* PlayAnim - Проигрывание анимации
* GoLeft - Бег вправо
* GoRight - Бег влево
* GoJump - Прыжок
* VelocityXY - Добавление скорости тела по ОсиX и ОсиY
* VelocityX - Добавление скорости по ОсиX
* VelocityY - Добавление скорости по ОсиY
* GetCollision - Возвращает векторную коллизию
* GetPositionX - Возвращает координату объекта по ОсиХ
* GetPositionY - Возвращает координату объекта по ОсиY
* SetPositionX - Присваивает координату объекта по ОсиХ
* SetPositionY - Присваивает координату объекта по
* MoveTo - Перемещение объекта
* GetLeft - Возвращает переменную, указывающую направление спрайта.

Выше перечисленные функции являются базовыми и будут использоваться во всех объектах движка. Поэтому они будут инициализированы в классе Object.cs.

Пример регистрации функции для скрипта Lua.

lua.RegisterFunction("GetCollision", this, typeof(Player).GetMethod("GetCollision"));

В первом параметре указывается название функции для скрипта. В последнем функция из кода которая должна вызываться.

Собственно обработка логики будет вызываться в методе Updateкласса Player.

lua.DoString("Main()");

Из-за того что класс предназначен и для игрока следовательно там описаны скрипт-функции нужные для игрока.

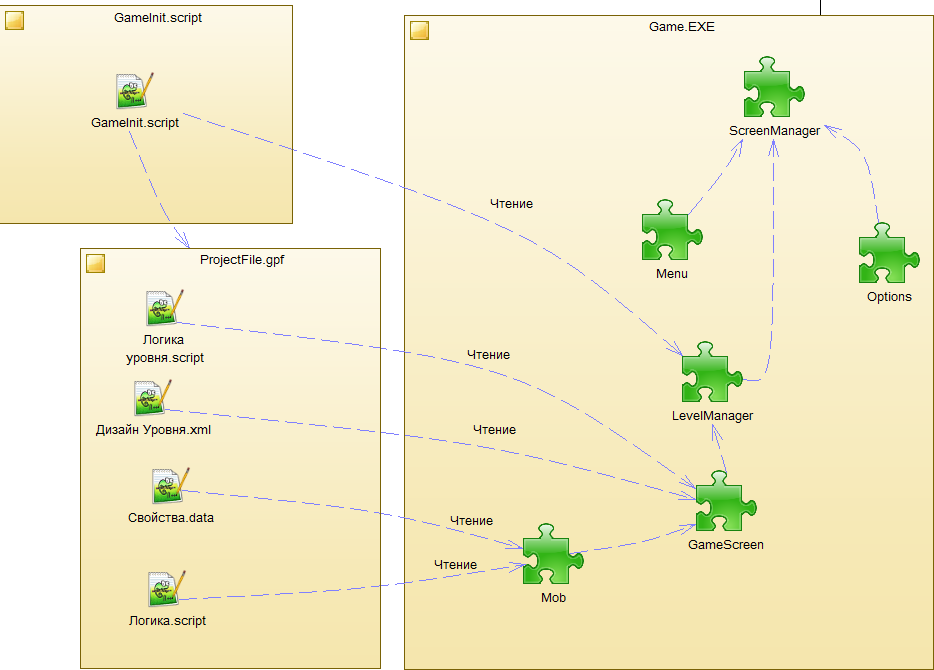
* ButtonPress
* ButtonDown

Данные функции предназначены для вычисления нажатия кнопки.

## 2.4. Разработка интерфейса конструктора

### 2.4.1 Разработка интерфейса

Поставленные требования к конструктору описанные в 2.1.3 является требованиям к интерфейсу. Интерфейс порождает проект файл(gpf) в котором храниться список скриптов, объектов, уровней и прикрепленных скриптов. Этот же проект файл прочитывается игровым движком.



**Рис 1.** Архитектура интерфейса

Скрипт к объекту прикрепляется способом Drag&Drop. Редактирование уровней реализуется в виде графического редактора.

Создание мобов есть в виде отдельной утилиты прикрепленный к интерфейсу, названный MobCreator. Данная утилита порождает отдельный data-файл. В этом файле будет храниться свойства моба и списки анимации с характеристиками, который свою очередь тоже прочитывается движком игры.

**Структура проект-файла**

* GameInit - главный запускающий игру файл
* путь к GameInit ..\GameInit.script.
* Число N указывающая на количество уровней
* Далее идут N уровни. Каждый уровень имеет свою структуру. Если один из структур уровня будет отсутствовать, то она пропускается.
  + Имя уровня
  + Путь к XML файлу
  + Скрипт привязанный к уровню(Сценарий уровня)
* Число K указывающая на количество мобов.
* Далее идут K мобов. Так же как и уровни имеет свою архитектуру.
  + Имя моба
  + Путь к data файлу моба
  + Путь к привязанному скрипту(Логика моба)
* Число L указывающая на количество скриптов
* Далее идут L скриптов. Так же как и уровни имеет свою архитектуру. Важно! Здесь GameInit.script тоже должен присутствовать.
  + Имя скрипта
  + Путь к файлу скрипта

Все пути относительны папки Application.

### 2.4.2 Интеграция GLEED2D к интерфейсу

**Разбор GLEED2D**

В конструкторе в качестве графического редактора используется готовая приложение под названием GLEED2D. GLEED2D (GenericLevelEditor 2D) является графическим редактором уровней, предназначенный для двумерных игр любого жанра. Лицензия: Microsoft Public License. Разработан с помощью XNA от компании Microsoft.

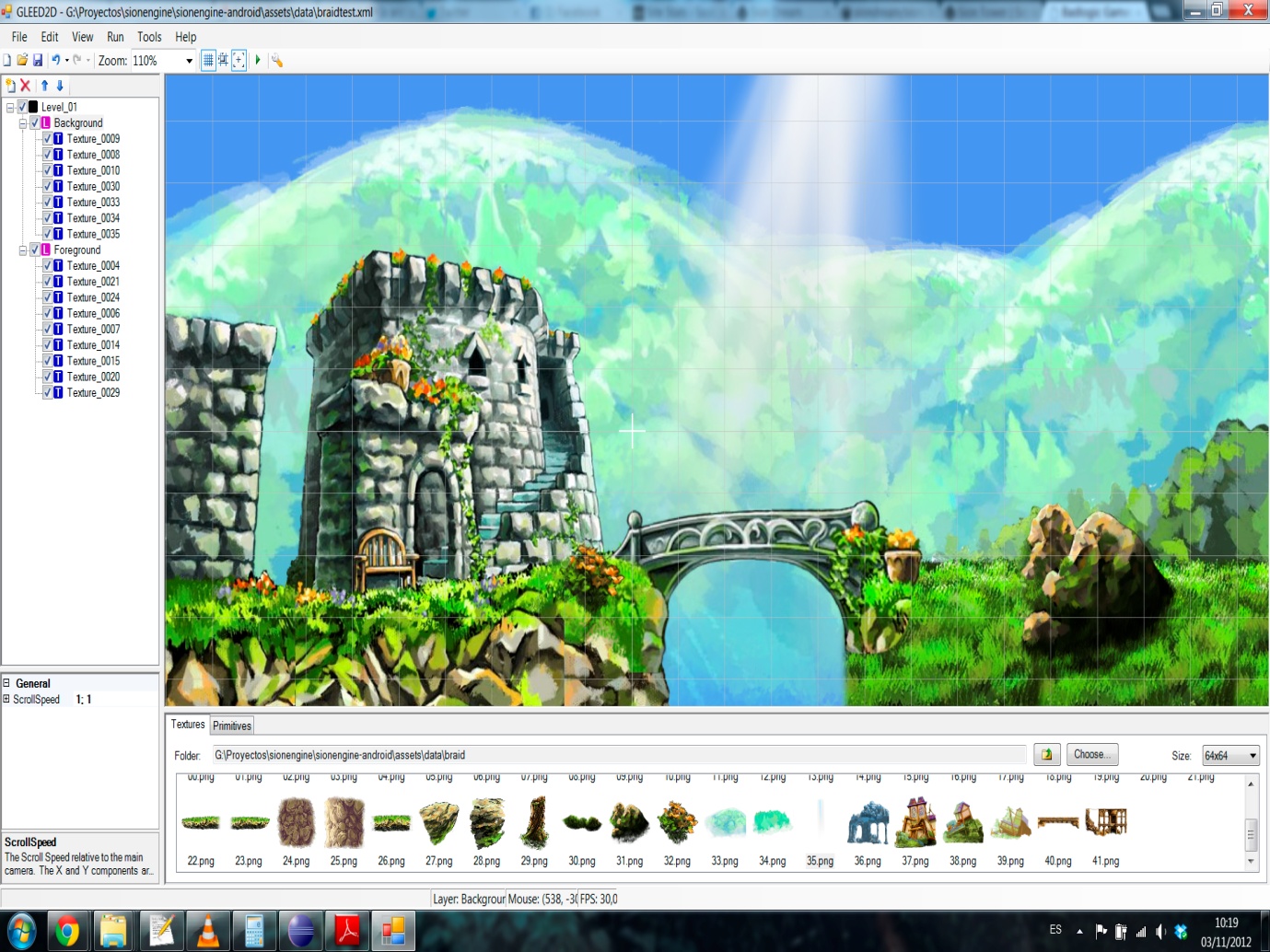


Рисунок 1. Скриншот GLEED2D

Состоит из одного компонента treeview где хранятся используемые текстуры. В каждой текстуре можно отредактировать свойства.

**Свойства:**

* Position

Координаты позиции текстуры в двумерном пространстве

* Origin

Координаты точка отсчета в текстуре

* Rotation

Угол поворота в радианах

* Scale

Масштаб объекта по Xи Y

* TintColor

Цвет окраски текстуры

* FlipHorizontally

Отразить текстуру по горизонтали

* FlipVertically

Отразить текстуру по вертикали

После расстановки текстур приложение порождает XML файл где храниться координаты текстур, свойства, путь к файлу текстуры и т.д.

Структура текстуры в XMLфайле.

<Item xsi:type="TextureItem" Name="Texture\_0001" Visible="true">

<Position>

<X>-2</X>

<Y>25</Y>

</Position>

<CustomProperties />

<Rotation>0</Rotation>

<Scale>

<X>20</X>

<Y>1</Y>

</Scale>

<TintColor>

<R>255</R>

<G>255</G>

<B>255</B>

<A>255</A>

<PackedValue>4294967295</PackedValue>

</TintColor>

<FlipHorizontally>false</FlipHorizontally>

<FlipVertically>false</FlipVertically> <texture\_filename>D:\ProgramFiles\Application\Content\Textures\Screens\Game\Level2\Platform.png </texture\_filename> <asset\_name>Content\Textures\Screens\Game\Level2\Platform</asset\_name>

<Origin>

<X>0</X>

<Y>30</Y>

</Origin>

</Item>

**Модификация GLEED2D**

Из требований описанных в главе 2.1.3 было решено добавить к GLEED2D следующий функционал:

* Сохранить/Загрузить игровой проект
* Создание игровых сцен
* Создание игровых объектов
* Создание и привязка к сценам или объектов скрипта.

Следовательно, интерфейс был модифицирован и дополнительно имеет списки сцен, объектов, скриптов, а также описание прикрепленного скрипта к объекту. Списки представлены как компоненты treeview.

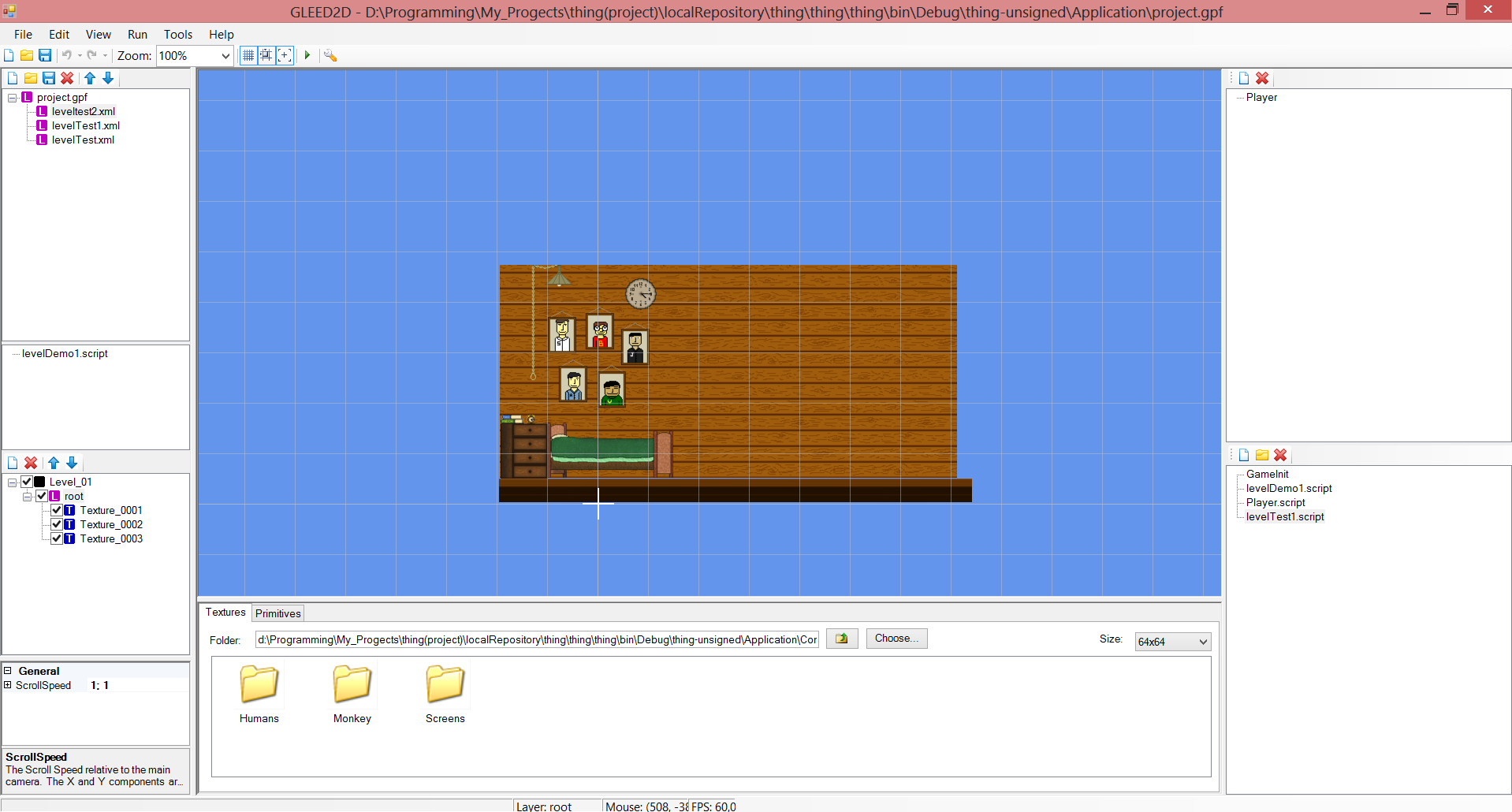


Рис. 3. Скриншот ннтерфейса конструктора

В первом списка, типа дерево, хранятся уровни. В левой нижней части хранятся список текстур в графической сцене. Этот список остался без изменений из оригинальной версии программы GLEED2D. В правой верхней части показывается список мобов. В правой нижней части показаны списки скриптов, созданные в рамках проекта модифицированной версии GLEED2D.

**Функции приложения модифицированные для реализации вышеописанного функционала**

Сперва были изменены существующие функции под требованный функционал, описанных в 2.1.3.

* private void FileSave(object sender, EventArgs e).

Изначально функция была предназначена для сохранения дизайна уровня и создания файла XML,структура которого описана в 2.4.2. При модификации было решено изменить его так чтобы он вызывал метод SaveProjectAs() и SaveProject(). Метод SaveProjectAs() вызывается когда требуется указать имя проекта, а SaveProject() при перезаписи текущего проекта.

* private void FileSaveAs(object sender, EventArgs e)

Предназначение функции была такая же как FileSave() за исключением того, что она требовала указать место сохранения и имя файла. После модификации сохраняет файл и записывает уровень в списке находящиеся верхней-левой части программы.

* private void FileOpen(object sender, EventArgs e)

Ранеезагружалуровень. ПарсилXMLфайливыводилуровень. Был переделан для загрузки проекта. Через OpenDialogвыбирается файл и начинется его обработка по образцу описанное в 2.4.1.

* public void export(string filename)

Ранее сохранял полный путь к файлу текстуры в XMLфайле. Был изменен и теперь сохраняет путь относительно папки Application. Это реализовано с целью сделать проект переносимым.

**Функции приложения разработанные для реализации вышеописанного функционала.**

Далее будут разобраны реализованные функции.

* private void SaveProject()

Функция, сохраняющая проект-файл по образцу, описанная в 2.4.1. Данные для сохранения берутся из списков.

* private void SaveProjectAs()

Через SaveFileDialog выбирается место для сохранения и имя проекта, после которого вызывается метод SaveProject().

* public void AddMob(MobCreator.Mob mob)

Метод вызываемый из утилиты MobCreator. Добавляет к дереву мобов параметры переданного класса mob.

Для хранения вершин списка был использован структура данных.

* private struct ObjectNodeTag

Это структур предназначена для хранения уровнейи мобов в дереве. Состоит из двух типов string. Первый предназначен для хранения пути к файлу XML или к файлу data.Второй для хранения пути к файлу скрипта.

Для хранения скрипта в дереве создание структуры не требовался. Так как он состоит из двух параметров, для хранения пути использовался поле Tagкласса TreeNode, а для хранения имени Text.

Для реализации Drag&Dropбыли использованы события DragDrop, DragEnterи DragOver.

В качестве редактора скрипта используется блокнот. При создании нового скрипта блокнотом открывается файл script с уже напечатанным архитектуро описанная в 2.3.7.

**MobCreator**

Данная программа является дополнением к модификации GLEED2D. Был написан на VisualStudio 2008 C#. Основной его функционал заключается в создании и настройке Мобов создаваемой игры, в результате, которого создает бинарный data файл.

Она состоит из двух частей.

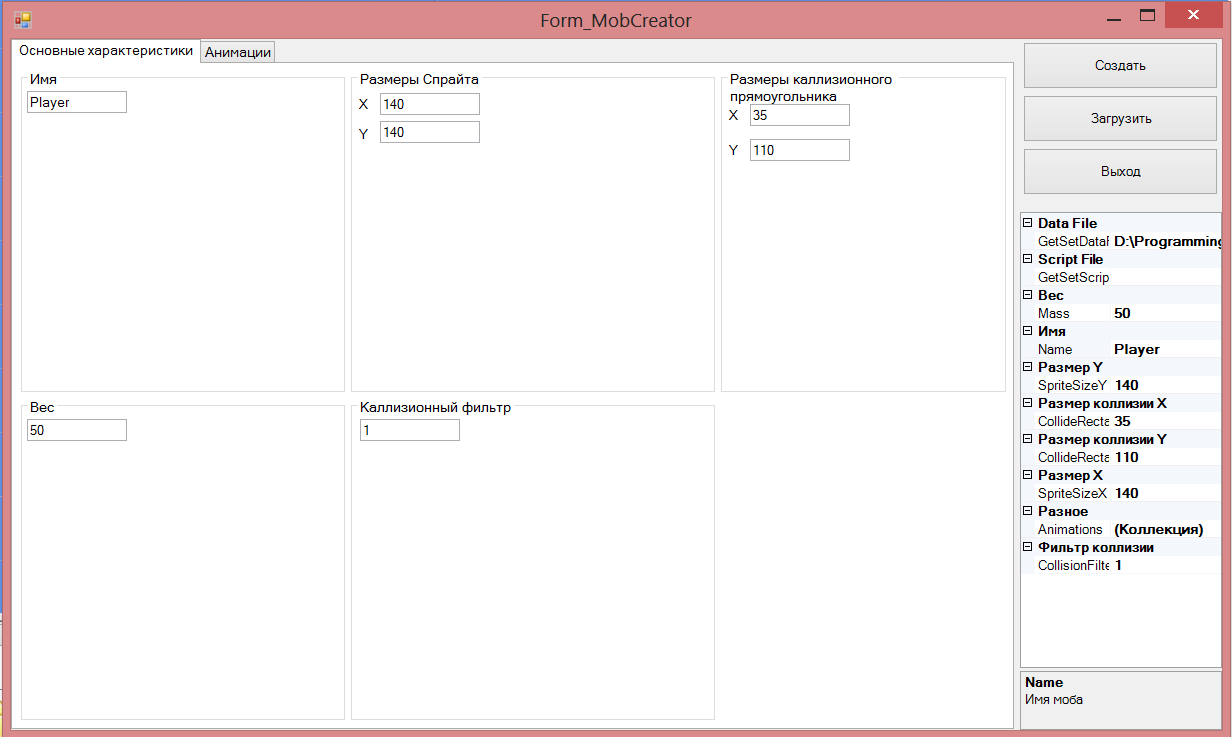


Рис. 4. Скриншот первой части утилиты MobCreator.

В первой части указываются основные свойства мобов. Например: Имя, размеры спрайта, размеры физического объекта и т.д.

Для этой реализации было разработан класс Mobгде хранились свойства, полученные из компонентов GUI(textbox).

**Mob**

Так как класс был разработан только для хранения в нем присутствует лишь один метод.

**Поля класса:**

* private List<Animation> animations = new List<Animation>();

Список из классов Animation. Здесь хранятся анимации который использует моб.

* private float collideRectangleSizeX;
* private float collideRectangleSizeY;

Поля для хранения размеров физического объекта моба.

* private int collisionFilter;

Поля указывающая на номер фильтра физической сцены

* private string dataFilePath = "";

Путь к data файлу.

* private float mass;

Вес физического объекта

* private string name;

Имя моба

* private string scriptFilePath = "";

Путь к файлу скрипта

* private float spriteSizeX;
* private float spriteSizeY;

Размер спрайта в графической сцене.

**Методы класса:**

* public void Print(BinaryWriter binWriter)

Метод создающийdataфайл.Он вызывается из интерфейса конструктора при сохранении проекта.

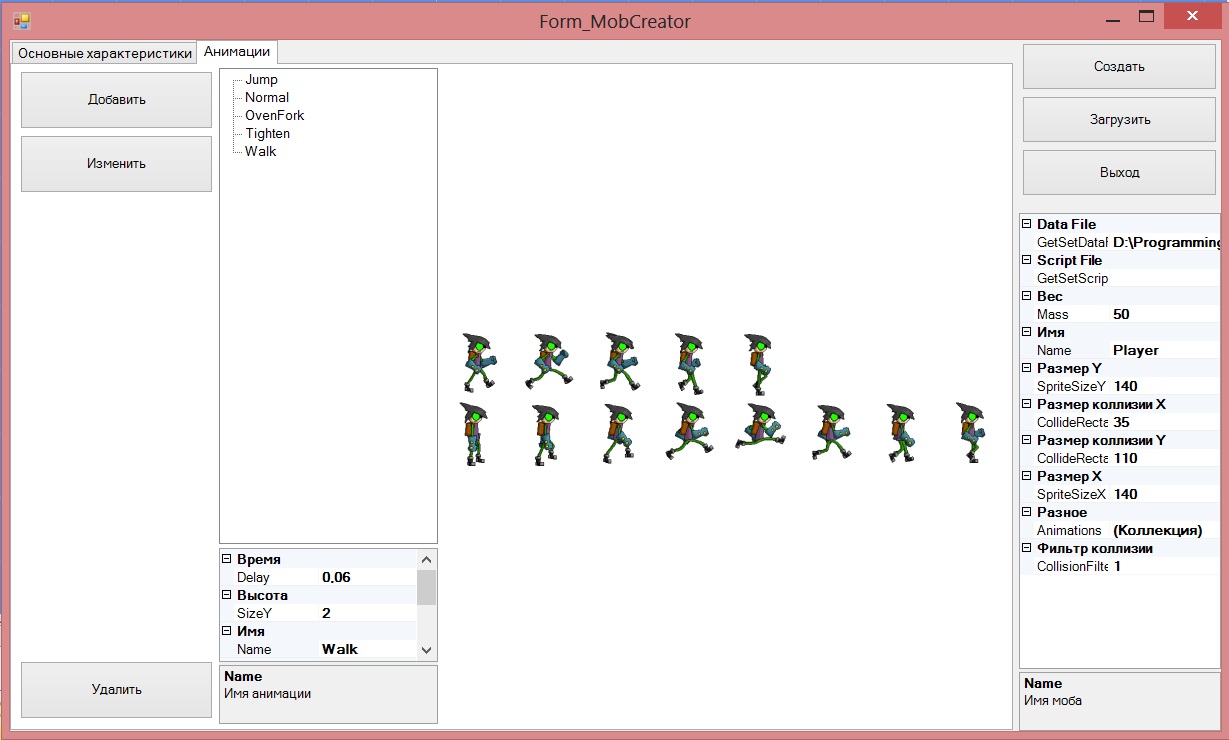


Рис. 5. Скриншот второй части утилиты MobCreator.

Вторая часть программы нацелен на создание и настройки анимаций мобов.

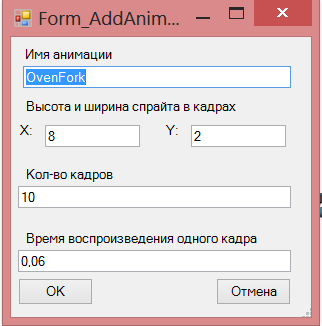


Рис. 6. Скриншот утилиты MobCreator где указывается свойства анимации.

Указываются такие данные как текстуры и свойства текстурам и свойства к анимациям.

Для него был разработан класс Animation.

**Animation**

Так же как и класс Mob. Этот класс предназначен для хранения свойств анимации. Входит в состав класса Mobв виде элементов списка.

**Поля класса**

* private float delay;

Время одного кадра.

* private int frameCount;

Указывает из скольких кадров состоит анимация

* private string fullPath;

Полный путь к файлу анимации(к спрайту)

* private string name;

Имя анимации

* private int sizeX;
* private int sizeY;

Ширина и высота спрайта в кадрах

После создание Mob нажав на кнопку сохранить вызываем метод AddMob() из интерфейса.

### 2.4.3 Интеграция игрового движка и интерфейса

Контент должен загружаться из папки Application. Проект файл и GameInit.script должны лежать в одном каталоге с исполняемым файлом (движком). Это обусловлено тем что в GameInit.script не записывается полный путь к файлу проекта.

В итоге интерфейс может находиться в другом месте от исполняемого файла движка.

## 2.5. Финальные тесты

В качестве финальных тестов было решение сделать простую игру жанра платформер.

Из-за ограничений, введенных в предыдущей подразделе, весь используемый контент был скопирован в папку Application.

Следующим шагом стало разработка уровня с помощью графического редактора(см. на рис 1).

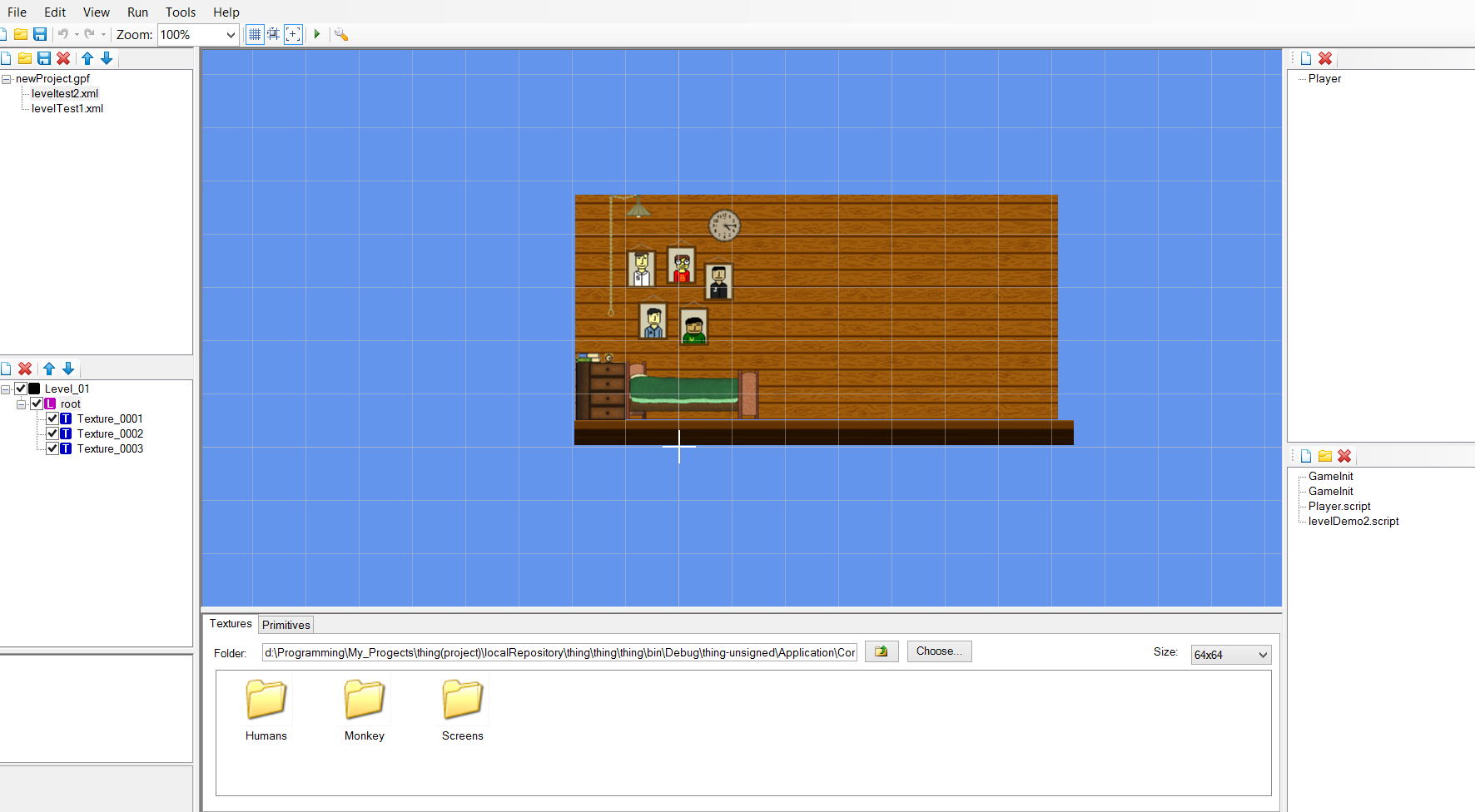


Рис. 1. Составление уровня

Потом был создан объект моб с помощью утилиты MobCreator.

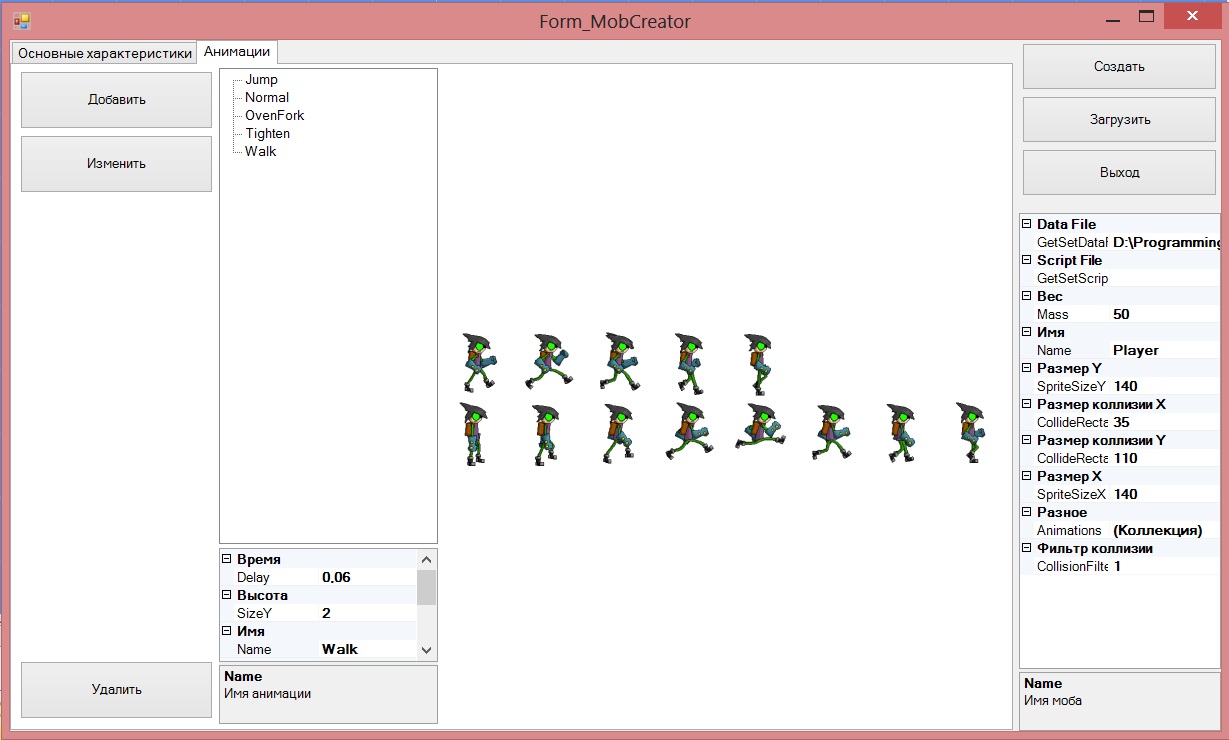


Рис. 2. Создание моба.

Создав моб, прописали к нему логику.

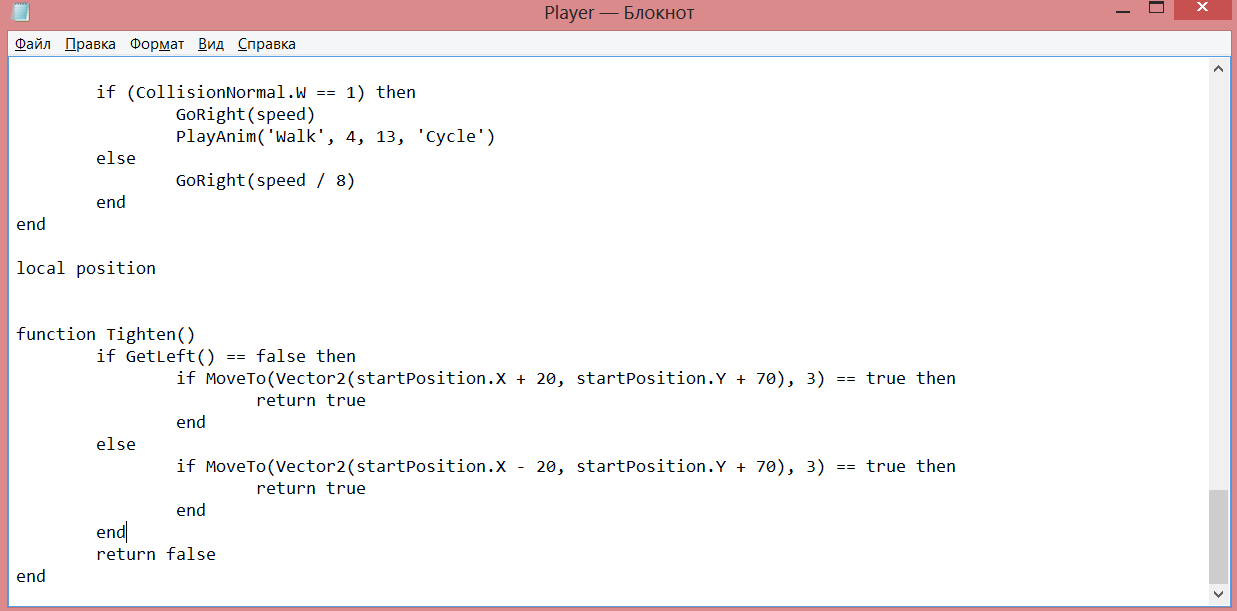


Рис. 3. Логика игрока написанного на Lua

Моб был инициализирован и его расстановка в сцене описан в скрипте:

function Initialize()

MobInit('Player', 140, 40, 1)

end

function Update(x, y)

end

Функция Updateосталась пустой, что означает нету сценария для этого уровня.(см 2.3.7)

Для первого запуска все необходимые компоненты были реализованы. А именно

* Уровень игры
* Инициализация уровня
* Моб (В данном случае игрок)
* Логика моба

Для запуска игры нужно написать скрипт в GameInit.script(см. 2.2)

projectName = ''

function Initialize()

projectName = 'newProject.gpf'

StartLevel('levelTest1')

end

Игра запустилось. Тест прошел успешно.

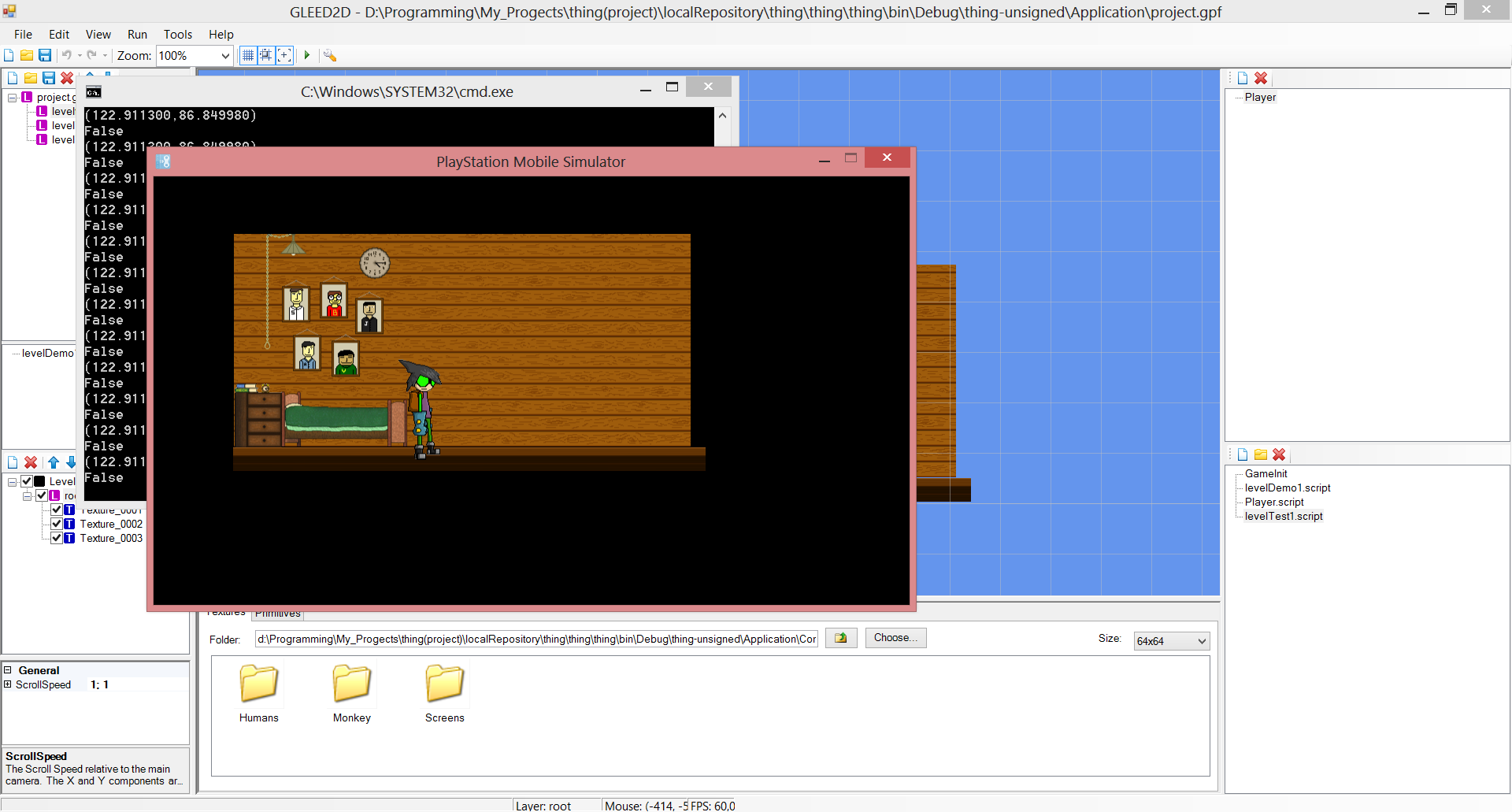


Рис. 4. Финальный результат

Итого для создания простого платформера в конструкторе было написано 143 строк кода Lua. Для сравнения возьмем код игры «Piercenfield» написанного на PSM Studio, который занимает 6996 строк кода.

# Заключение

Цель дипломной работы – разработать конструктор двумерных компьютерных игр на платформе PSM, позволяющая пользователям работать только в плане геймплея (игрового процеса) игры. Для достижения данной цели были решены следующие задачи:

1. Изучить существующие конструкторы и разработать архитектуру для собственного конструктора
2. Изучить существующие игровые движки
3. Разработать игровой движок, его архитектуру и реализовать его компоненты
4. Разработать интерфейс конструктора
5. Объединить интерфейс и игровой движок получив единую систему конструктора
6. Протестировать полученный код

Для реализации первой задачи были изучены уже существующие конструкторы игр. На основе этого были поставлены требования, которые нужны к простейшему конструктору.

Для реализации второй задачи также были рассмотрены существующие игровые движки. Благодаря этому было изучено, из каких компонентов состоят игровые движки. К изученным компонентам были поставлены требования и цели для их реализации.

Реализации третьей части послужил поиск готовых графических и физических движков. Для графического движка был разработан дополнительный компонент для вывода анимации. К физическому движку был разработан интерфейс для легкого доступа к объектам движка. Реализацией поставленных требований стала разработкой объектов, предоставляемых движком.

Для реализации четвертой задачи нужно было модифицировать готовую программу GLEED2D. Были добавлены такие возможности как:

* Сохранение/загрузка проектов.
* Создание и загрузка сцен, мобов и скриптов.
* Дополнительная программа для создания и настройки мобов.

Результатом реализации пятой задачи стало появление некоторых ограничений. Проект-файл создаваемый интерфейсом должен храниться в теле игры, в папке Application. Текстуры, скрипты и объекты, используемые в создаваемой игре должны храниться и загружаться в интерфейсе, в папке Application.

Результатом шестой задачи стало создание игры-платформера, созданной с помощью созданного конструктора.

Таким образом, поставленные задачи решены, и цель работы достигнута.

# Список использованной литературы

1. **Корпорация Nintendo.**Consolidated Sales Transition by Region. *Investor Relations.* [В Интернете] 24 04 2013 r. [Цитировано: 08 06 2013 r.] http://www.nintendo.co.jp/ir/library/historical\_data/pdf/consolidated\_sales\_e1303.pdf.

2. **Microsoft.** Earnings Release FY13 Q2. 15 февраля 2013. Проверено 2013-01-24.mdy.2013. [В Интернете] 15 02 2013 r. [Цитировано: 08 06 2013 r.] http://www.microsoft.com/investor/EarningsAndFinancials/Earnings/Kpi/fy13/q2/Detail.aspx.

3. **Sony Computer Entertainment.** PlayStation®3 Sales Reach 70 Million Units Worldwide. *sony.com.* [В Интернете] 16 11 2012 r. [Цитировано: 08 06 2013 r.] http://www.sony.com/SCA/company-news/press-releases/sony-computer-entertainment-america-inc/2012/playstation3-sales-reach-70-million-units-worldwid.shtml.

4. **Александр Лобао, Бруно Евангелиста, Жозе Антонио Лил Де Фариас.***Основы программирования игр с XNA 2.0. От новичка до профессионала.* 2008.

5. **Holman, Paul.**PlayStation Development: An Introduction. *SCEE Research and Development.* [В Интернете] 2012 r. http://develop.scee.net/files/presentations/jordan2012/2012-Jordan%20Game%20Summit-PlayStation%20Mobile.pdf.

6. **Фленов, Михаил.***Исскуство программирования игр на С++.* Санкт-Петербург : БХВ-Петербург, 2006.

7. **Stuart, Keith.**Stuart, Keith. *3D games enter a new generation.* [В Интернете] 19 Сентябрь 2010 r. [Цитировано: 17 06 2014 r.] http://www.guardian.co.uk/technology/2010/sep/19/3d-games-xbox-playstation.

8. **Specusphere.** "Gamespeak: A glossary of gaming terms". [В Интернете] 21 01 2007 r. [Цитировано: 06 17 2014 r.] http://wayback.archive.org/web/20070219082328/http://www.specusphere.com/joomla/index.php?option=com\_content&task=view&id=232&Itemid=32.

9. Термины и определения в программировании игр . [В Интернете] [Цитировано: 17 06 2014 r.] http://ekaterinodar.ru/talk/razrabotka\_igry/topics/3507.html.

10. **Р., Сузи.** Сценарные языки: Python. *«Мир ПК».* [В Интернете] 9 2001 r. [Цитировано: 17 06 2014 r.]

11. **HITBOXTEAM.**dustforce. *dustforce.* [В Интернете] [Цитировано: 2014 06 17 r.] http://dustforce.com/about.

12. **inXile enterainmed.** Wasteland. *Wasteland.* [В Интернете] 2014 r. [Цитировано: 17 06 2014 r.] http://wasteland.inxile-entertainment.com/.

13. **Sony Computer Entertainment.** [В Интернете] [Цитировано: 29 11 2013 r.] https://mail.rambler.ru/m/redirect?url=https%3A//psm.playstation.net/static/general/all/sdk\_eula.html&hash=775dd7923ebc3d848990afae6afe5fac.

14. **Elecbyte.**MUGEN. *Elecbyte.* [В Интернете] [Цитировано: 18 06 2014 r.] http://elecbyte.com/.

15. **Cocos2D.**index. *Cocos2D.* [В Интернете] 29 03 2008 r. [Цитировано: 08 06 1992 r.] http://www.cocos2d.org/.

16. **MonoGame.** About. *MonoGame: Write Once, Play Everywhere.* [В Интернете] [Цитировано: 08 06 2013 r.] http://www.monogame.net/.

17. **Red Thread Games.** Dreamfall chapters: the longes journey. *Dreamfall.* [В Интернете] 1 3 2007 r. [Цитировано: 17 06 2014 r.] http://www.dreamfallchapters.com/.