**Содержание**

[**Содержание** 3](#_Toc421263788)

[Введение 4](#_Toc421263789)

[1 Обзор существующих инструментов 6](#_Toc421263790)

[1.1 Общие сведения 6](#_Toc421263791)

[1.2 Типы вершин 8](#_Toc421263792)

[1.2.1 Лист 8](#_Toc421263793)

[1.2.2 Композит 8](#_Toc421263794)

[1.2.3 Декоратор 10](#_Toc421263795)

[1.3 Деревья поведения в современных фреймворках 11](#_Toc421263796)

[1.3.1 Unreal Engine 11](#_Toc421263797)

[1.3.2 LibGDX 13](#_Toc421263798)

[1.3.3 Unity3D 14](#_Toc421263799)

[2 Раздел 2 16](#_Toc421263800)

[2.1 Подраздел 1 16](#_Toc421263801)

[2.2 Подраздел 2 16](#_Toc421263802)

[Заключение 17](#_Toc421263803)

[Список использованных источников 18](#_Toc421263804)

[Приложение 19](#_Toc421263805)

[Приложение 21](#_Toc421263806)

[Приложение 22](#_Toc421263807)

[Приложение Объемы рынка игр 23](#_Toc421263808)

Введение

Объем рынка игр растет с каждым годом (смотри рисунок Г.1 и Г.2). В первую очередь развитие игровых приложений обязано стремительному прогрессу, но также важная роль в разработке игр ложится на плечи программистов. Именно с их помощью реализуются идеи геймдизайнеров, именно они собирают воедино воображения художников с целью создания новой игры.

Программисту, в настоящее время, чтобы написать игру, что называется, «с нуля» необходимо позаботиться о многих важных аспектах: игровая платформа, графическая система, аудио система, системы моделирования игровой физики и виртуального интеллекта игры. Каждая из этих областей объемна и потребуется много времени, чтобы связать воедино все системы в соответствии с игровой логикой.

Для того, чтобы упростить и ускорить работу над созданием игры программисты написали множество библиотек для работы с графическими системами и для моделирования игровой физики. Такие библиотеки, как Lightweight Java Game Library, предоставляют инструменты для работы с графической и аудио системами, а библиотека box2d предоставляет инструменты для моделирования реалистичной физики в плоском пространстве. Однако для упрощения реализации виртуального интеллекта долгое время не было придумано, по большому счету, ничего. Над созданием искусственного интеллекта в игре программист трудился самостоятельно, так как этот процесс не был сложным или утомительным. В конечном итоге модуль виртуального интеллекта содержал в себе некоторое количество условных операторов.

Такое решение обнажило бы свою слабую сторону в играх с развитым поведением персонажей потому, что условных операторов становилось очень много, что затрудняло отладку и дальнейшую поддержку игры. Пример такой игры – Spore. В ней все игровые персонажи имели если не уникальное, то редко повторяющееся поведение, для создания которого в Spore использовался совершенно другой подход.

Только в последнее десятилетие (примерно с момента выхода игры Spore) виртуальный интеллект в играх стал развиваться и появились некоторые библиотеки AIEngine (Artificial Intelligence Engine), которые обобщали накопленные программистами знания об интеллекте в играх. Эти библиотеки предоставляют инструменты для создания конечных автоматов поведения, для обработки взаимодействия автономных объектов. Но конечные автоматы в общем виде сложны и часто запутаны. Чтобы стратегия объекта была ясной были созданы деревья поведения – конечные автоматы древовидной структуры, состоящие из вершин трех типов: вершины-действия, вершины-условия и управляющие вершины. Вершина-действие содержит в себе некоторое возможное действие объекта (бежать, искать, стрелять), вершина-условие содержит в себе некоторый предикат (есть патроны?, враг рядом?), в зависимости от которого выбирается следующее состояние, управляющая вершина организует порядок обхода дочерних вершин (параллельно, до первой успешной вершины, до первой неуспешной вершины).[[1]](#footnote-1)

Цель данной работы - проектирование библиотеки и реализация визуального средства для создания стратегий поведения виртуальных игровых персонажей на основе деревьев поведений.

1. Обзор существующих инструментов

## Общие сведения

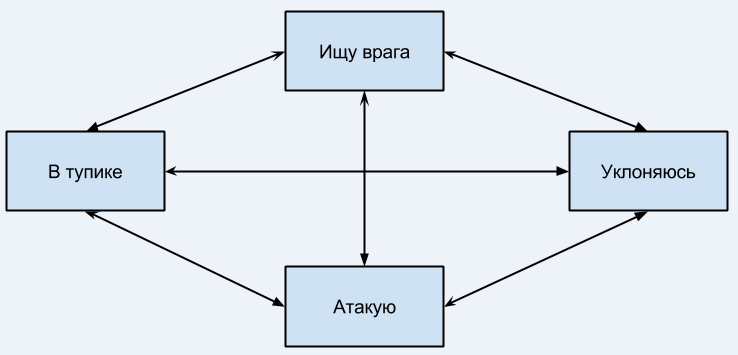
Для разработки виртуальных игровых объектов необходимо создать и поддерживать большой набор их поведений. Например, в военных играх виртуальному персонажу необходимо распознавать опасность и убегать в укрытие, когда уровень здоровья ниже 10%. От количества разнообразных вариантов действий, которые могут использовать игровые персонажи, зависит количество игровых ситуаций, которые могут быть распознаны и приняты во внимание. Чем больше различного поведения игроки будут встречать в играх даже от несущественных (фоновых) объектов[[2]](#footnote-2), тем интереснее будет игра. 

Рисунок 1.1 - Конечный автомат поведения игрового объекта в военных играх

Для создания поведения долгое время использовались конечные автоматы, где каждое поведение может быть представлено графически (рисунок 1.1). Одновременно автомат может находиться в одном состоянии, которое представляет поведение объекта. Каждое состояние имеет логику переходов с соответствующими проверками. Например, если игрок находится в состоянии «В тупике» и обнаружил врага, и здоровье игрока больше 50%, то следующим состоянием будет «Атакую».

Такой подход к реализации принятия решений игровыми объектами имеет ряд недостатков:

* Расширяемость: конечный автомат с большим количеством состояний теряет преимущество графического представления, со временем такое поведение станет невозможно понять.
* Изменяемость: при добавлении/удалении поведения (состояния) необходимо изменить все другие состояние, которые связаны с новым/старым поведением; большие изменения могут приводить к ошибкам логики поведения объекта в целом.
* Параллельные алгоритмы: запускать состояния автомата параллельно не представляется возможным.

В 2005 году был создан более эффективный способ принятия решения игровыми объектами по сравнению с конечными автоматами – деревья поведения.

Дерево поведения – это направленный связный ациклический[[3]](#footnote-3) граф, имеющий единственную вершину, в которую не входят ребра – корень дерева. Из пары вершин, соединенных ребром, та, из которой выходит ребро, называется родительской вершиной, а другая дочерней вершиной. Вершина, не имеющая дочерних, называются листом. Каждое поддерево дерева поведения определяет различное поведение. Вершины, находящиеся между корнем дерева и листами могут быть двух типов – декораторами или композитами. Корень дерева поведения периодично генерирует сигнал, который передает дочерним вершинам, заставляя их выполнять алгоритм, определенный типом вершины. Как только сигнал достигнет листа, то лист произведет некоторые вычисления и вернет одно из 4 состояний: «успешно» (success), «не успешно» (failure), «запущено» (running), «ошибка» (error). Возвращенное состояние передастся родительским вершинам, для принятия решений в соответствии с типом вершины. Процесс закончится тогда, когда корневая вершина вернет некоторое состояние.

## Типы вершин

Все вершины дерева поведения делятся на три типа: декоратор, композит, лист.

## Лист

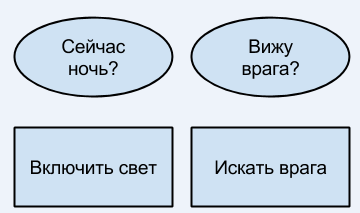
Лист – это не делимая часть дерева поведения. Эти вершины не имеют дочерних вершин, они принимают сигнал, производят некоторые вычисления и возвращают результат родительской вершине. 

Рисунок 1.2 - Графическое представление листовых вершин

Существует два вида листовых вершин: лист-условие и лист-действие. Лист-условие выполняет проверку некоторого условия и возвращает соответствующий результат (успешно, не успешно или ошибка). Лист-действие выполняет действие и возвращает результат успешно, запущено или ошибка.

Графически лист-условие изображается овалом, лист-действие прямоугольником (рисунок 1.2).

## Композит

Композит имеет одну или больше дочерних вершин. Он принимает и передает сигнал дочерним вершинам в некотором порядке, и также решает какое и когда вернуть состояние. Композит всегда возвращает одно из трех состояний: «успешно», «не успешно» или «ошибка». Все композитные вершины изображаются в виде квадрата со специальным символом внутри.

Существует три вида композитной вершины (рисунок 1.3): композит-селектор, композит-последовательность и параллельный композит. Композит-селектор обрабатывает дочерние вершины до тех пор, пока дочерняя вершина возвращает результат «не успешно», затем пробрасывает полученный результат родительской вершине и заканчивает выполнение. Если все дочерние вершины вернули результат «не успешно», то композит-селектор вернет результат «не успешно». Специальный символ для композита-селектора – знак вопроса.

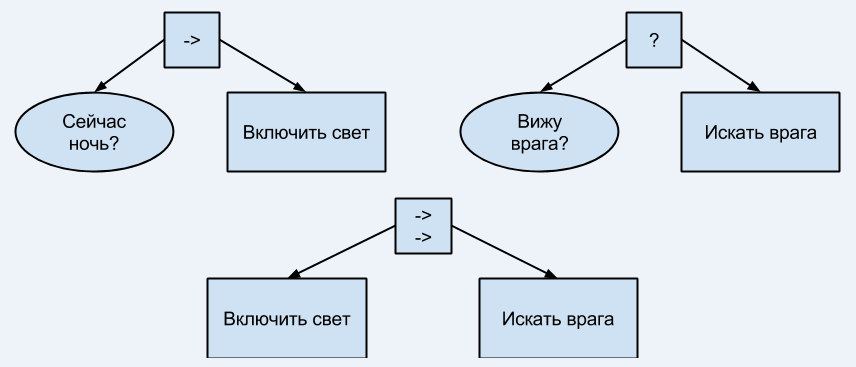
Композит-последовательность обрабатывает дочерние вершины до тех пор, пока они возвращают результат «успешно», затем пробрасывает полученный результат родительской вершине и заканчивает выполнение. Если все дочерние вершины вернули результат «успешно», то композит-последовательность вернет результат «успешно». Специальный символ для композита-последовательности – стрелка вправо.

Рисунок 1.3 – Примеры вершин типа «композит»

Параллельный композит обрабатывает все вершины одновременно, возвращает «успешно», если количество дочерних вершин с результатом «успешно» превышает некоторую константу S (которая может быть различна для разных параллельных композитов), возвращает результат «не успешно», если количество дочерних вершин с результатом «не успешно» превышает некоторую константу F, которая так же может быть определена для конкретного параллельного композита, иначе возвращает результат «запущено». Специальный символ для параллельного композита – две стрелки вправо.

## Декоратор

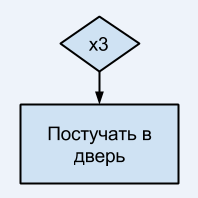
Декоратор – это специальная вершина, которая имеет ровно одну дочернюю вершину. Цель, которую преследует декоратор, - изменить возвращаемое дочерней вершиной значение, или повлиять на частоту передаваемого сигнала дочерней вершины. Например, декоратор может делать инверсию возвращаемого значения, а может повторить сигнал, передаваемый дочерней вершине 3 раза. Декоратор изображается в виде ромба с пояснением внутри.

Рисунок 1.4 – Пример вершины-декоратора, который три раза передаст сигнал дочерней вершине

## Взаимодействие различных вершин

Автономным объектам необходимо в процессе принятия решения хранить промежуточную информацию об окружающем мире. Эта информация формирует систему «восприятия» мира для объекта, она может включать, например, последнюю видимую позицию врага, количество видимых объектов, последнее совершенное действие или любые другие вычисленные данные. Таким образом дерево поведения некоторого игрового объекта должно хранить и использовать некоторую информацию о мире.

Для решения данной задачи применяется blackboard [3] (рисунок 1.5), который активно используется вершинами дерева поведения для чтения и записи информации. Blackboard – это ассоциативный массив, к которому имеют доступ все вершины дерева поведения.

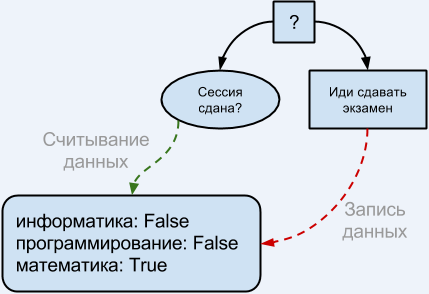


Рисунок 1.5 - Использование пула памяти

## Деревья поведения в современных фреймворках

В некоторых достаточно крупных игровых фреймворках существует возможность создания стратегии персонажей с помощью деревьев поведения. Мы рассмотрим три фреймворка, использующих эту технологию: Unreal Engine, LibGDX, Unity3D.

## Unreal Engine

UnrealEngine – игровой движок, разрабатываемый компанией Epic Games. Различные версии этого фреймворка были использованы во многих современных играх.

Для реализации стратегии для автономных объектов в UnrealEngine можно использовать деревья поведения [4], причем в данном игровом движке есть возможность создать дерево поведения, используя графический интерфейс, при этом не написав ни строчки кода.

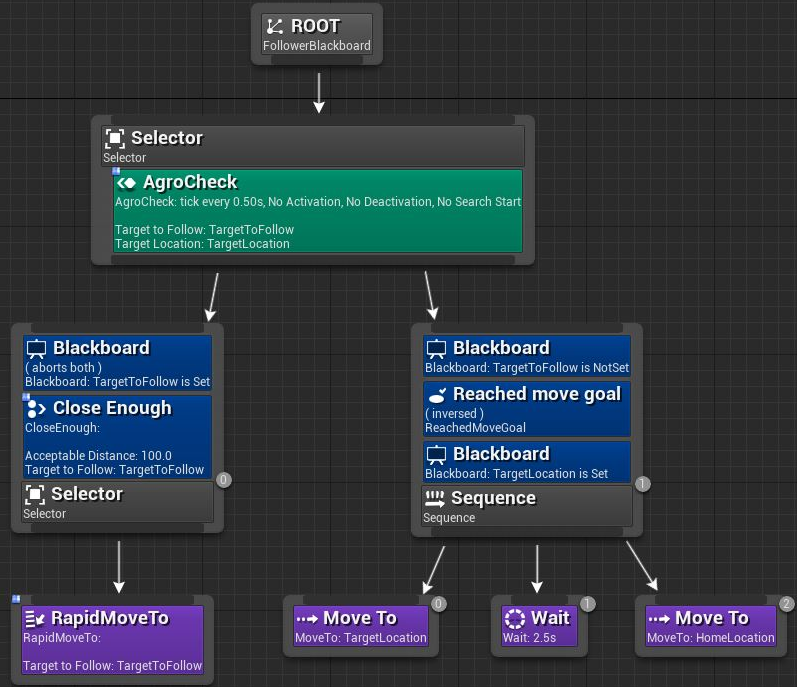
Рассмотрим использование деревьев поведения в UnrealEngine на примере: имеется игровой мир в виде комнаты со стенами, в котором находятся два персонажа, один из них – человек, другой – виртуальный интеллект, задача виртуального интеллекта – найти человека и подойти к нему. Для реализации такого рода стратегии в UnrealEngine необходимо построить дерево поведения (смотри рисунок 1.6).

Рисунок 1.6 - Реализация дерева поведения в UnrealEngine

Данное дерево состоит из двух вершин-селекторов, вершины-последовательности, четырех вершин-действий. В первой вершине-селектор находится дополнительное действие «AgroCheck», которое реализует «зрение» виртуального интеллекта, устанавливая, в частности, две переменные: позиция и ссылка на персонажа, которого необходимо отыскать. Композит-селектор слева включает в себя два декоратора, которые проверяют установлена ли ссылка на персонажа и приемлема ли до него дистанция, при верном ответе на эти два вопроса выполняется действие – быстрое движение к искомому объекту. Если же объект пропадет из поля видимости, то тогда сигнал пойдет к левому композиту-последовательности, который содержит три декоратора, проверяющих, что цель не установлена, цель не достигнута и некоторая позиция установлена, в этом случае будут выполняться следующие действия. Сначала виртуальный интеллект переместится до установленной позиции (скорее всего это та позиция, где он последний раз видел другого персонажа), затем подождет 2.5 секунды и отправится на исходную точку.

## LibGDX

LibGDX – кроссплатформенный игровой фреймворк, написанный на языке Java и C++. LibGDX используют для написания мобильных приложений и игр. В своем составе этот фреймворк имеет модуль AI [5], который реализует алгоритмы нахождения кратчайших путей, взаимодействия автономных объектов (steering behavior), а также алгоритмы принятия решений на основе behavior tree.

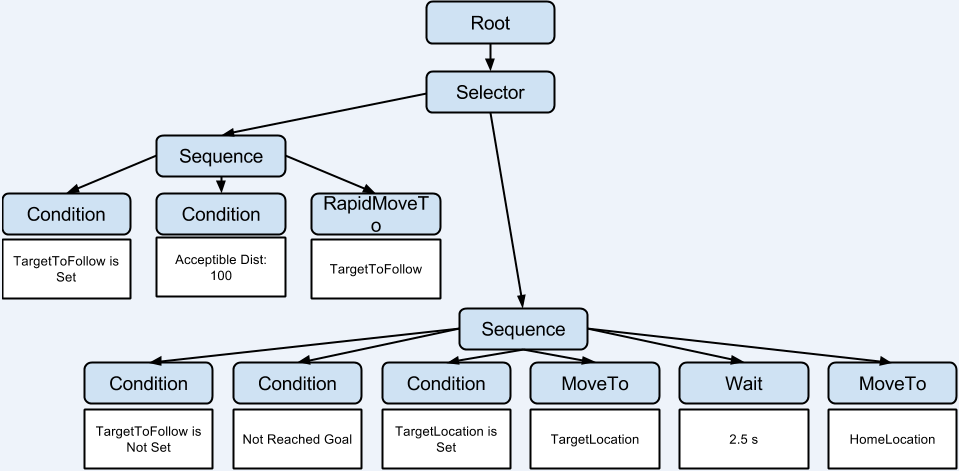
Рассмотрим создание дерева поведения на том же примере, что и в предыдущем разделе. Так как в деревьях поведения LibGDX AI нет декораторов, проверяющих некоторое условие, то аналогичное дерево поведения будет выглядеть как показано на рисунке 1.7.

Рисунок 1.7 - Дерево поведения с использованием модуля LibGDX AI

Так как в LibGDX AI не предусмотрено графического средства для создания деревьев поведения, то полученное на рисунке 1.7 дерево необходимо создать непосредственно в коде.

private BehaviorTree<Blackboard> createHeroBehavior() {

ConditionTask<Blackboard> targetToFollowIsSet = new ConditionTask<>(

bb -> bb.targetToFollow != null);

ConditionTask<Blackboard> acceptibleDist = new ConditionTask<>(

bb -> dist(bb.targetToFollow.location, me.location) <= 100);

RapidMoveTo<Blackboard> rapidMoveTo =

new RapidMoveTo<>(bb.targetToFollow.location);

Sequence<Blackboard> sequenceSeeMan = new Sequence<>(

targetToFollowIsSet, acceptibleDist, rapidMoveTo);

ConditionTask<Blackboard> targetToFollowIsNotSet = new ConditionTask<>(

bb -> bb.targetToFollow == null);

ConditionTask<Blackboard> notreachedGoal = new ConditionTask<>(

bb -> dist(bb.targetLocation, me.location) > 1);

ConditionTask<Blackboard> targetLocationIsSet = new ConditionTask<>(

bb -> bb.targetLocation != null);

MoveTo<Blackboard> moveToTargetLocation = new MoveTo<>(bb.targetLocation);

Wait<Blackboard> wait = new Wait<>(2500);

MoveTo<Blackboard> moveToHomeLocation = new MoveTo<>(bb.homeLocation);

Sequence<Blackboard> sequenceNotSeeMan = new Sequence<>(

targetToFollowIsNotSet,

notreachedGoal,

targetLocationIsSet,

moveToTargetLocation,

wait,

moveToHomeLocation

);

Selector<Blackboard> selector = new Selector<>(sequenceSeeMan, sequenceNotSeeMan);

Selector<Blackboard> root = new Selector<Blackboard>(selector);

BehaviorTree<Blackboard> bt = new BehaviorTree<>(root, blackboard);

return bt;

}

В LibGDX для каждой вершины дерева поведения есть общий класс – Task. Наследуясь от него можно создавать пользовательские типы вершин, которые не предусмотрены в базовой структуре классов. Так, лист-условие Condition, листы-действия RapidMoveTo, MoveTo и Wait необходимо создать и реализовать их логику. Так как композит-последовательность передает сигнал на исполнение дочерним вершинами до тех пор, пока они возвращают «успешно», то данное дерево будет аналогичным дереву в разделе 1.3.1. Действительно, если некоторый лист-действие вернет «не успешно», то композит-селектор прекратит передачу сигнала на исполнение и следующим дочерним вершинам сигнал не будет передан.

В методе createHeroBehavior описан процесс создания дерева поведения с использованием LibGDX AI. Лист-условие Condition задается предикатом с одним параметром – blackboard, с помощью которого и определяется собственно истинность условия. Листы-действия RapidMoveTo и MoveTo задаются с помощью функции, которая возвращает позицию, к которой необходимо двигаться. Лист-действие Wait задается длительностью интервала задержки в миллисекундах.

## Unity3D

Unity3D – игровой движок, разрабатываемый компанией Unity Technologies. В силу наличия бесплатной версии и огромного количества поддерживаемых платформ [7] этот движок весьма популярен среди многих крупных разработчиков игр (Blizzard, EA, QuartSoft, Ubisoft).

В составе Unity3D отсутствуют инструменты для работы с искусственным интеллектом, но в магазине плагинов [8] можно найти дополнение, позволяющее создавать деревья поведения для игровых объектов. Один из таких плагинов – Behaviour Machine Free. Рассмотрим создание дерева поведения с использованием данного плагина на примере из предыдущего раздела.

В Behaviour Machine Free много стандартных вершин, таких как Translate (переместить объект по указанному направлению на указанную длину), IsFloatLess/IsFloatGreater (проверить, что некоторое число с плавающей точкой меньше/больше заданного числа), GetDistance (получить расстояние между двумя объектами и записать результат в blackboard), IsSee (проверить, видит один объект другой или нет).

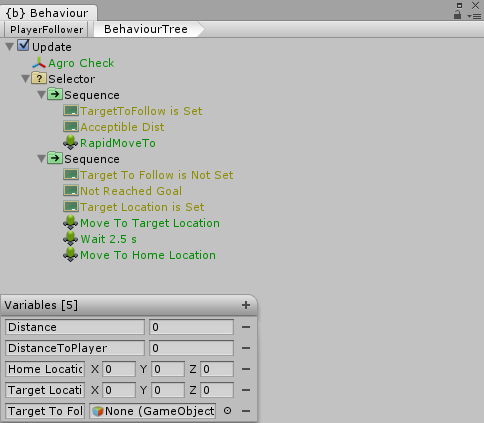
Таким образом, для того чтобы создать необходимое поведение для персонажа, нужно в графическом интерфейсе для определения способа принятия решений создать структуру дерева поведения и структуру blackboard (рисунок 1.8), а затем задать параметры для каждой вершины дерева. Так, для Agro Check необходимо задать объекты, между которыми нужно проверять видимость, для Acceptible Dist необходимо задать переменную и число, чтобы проверять, что переменная меньше числа, для Wait необходимо задать количество миллисекунд задержки и т.д.

Рисунок 1.8 - Дерево поведения с использованием Behaviour Machines на Unity3D

## Итоги

Таким образом, мы рассмотрели три крупных игровых фреймворка, в которых есть возможность использовать деревья поведения для создания логики принятия решений виртуальным интеллектом, и решили одну задачу с помощью этих фреймворков. Были отмечены следующие недостатки: использовать деревья поведения UnrealEngine или Behaviour Machine Free в проектах, написанных на других фреймворках не представляется возможным, а для модуля Behavior Tree из LibGDX AI не существует визуального средства создания деревьев поведения.

## Постановка задачи

Необходимо разработать кроссплатформенную библиотеку и визуальное средство для создания деревьев поведения.

1. Раздел 2

## Подраздел 1

## Подраздел 2

Для точного определения числа математики используют формулу Валлиса:

|  |  |
| --- | --- |
| , | (3.2) |

где n – целое число.

Сам Валлис пришел к этой формуле, вычисляя площадь круга. Это произведение сходится крайне медленно, поэтому для практического вычисления числа формула Валлиса не пригодна. Однако она полезна в теоритических исследованиях, например при выводе формулы Муавра – Стирлинга.

Заключение

Заключение должно содержать краткую характеристику результатов выполненной работы (результатов решения поставленных задач), и рекомендации по улучшению системы.

Список использованных источников

1. Towards a Unified Behavior Trees Framework for Robot Control / A. Marzinotto [и др.] .- Swedish Research Council and the European Union Project, 2013 .- 8 с.
2. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees / P. Ogren .- Swedish Defence Research Agency, Stockholm, 2012 .- 8 с.
3. Behavior Trees for Hierarchical RTS AI / S. Delmer .- Plano: The Guildhall at SMU, 2013.- 10 c.
4. Unreal Engine 4 Documentation - <https://docs.unrealengine.com/latest/INT/>
5. LibGDX Documentation - <https://github.com/libgdx/libgdx/wiki>
6. Unity3D Documentation - <http://docs.unity3d.com/Manual/index.html>
7. Unity3D site – <https://unity3d.com>
8. Unity3D Asset Store - <https://www.assetstore.unity3d.com>

**Задание на выполнение бакалаврской работы**

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

«Алтайский государственный технический университет им. И. И. Ползунова»

УТВЕРЖДАЮ

Заведующий кафедрой ПМ

\_\_\_\_\_\_\_\_\_ Кантор С.А.

*подпись ФИО*

**ЗАДАНИЕ №**

**НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ**

по направлению подготовки\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

по профилю\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

студенту группы Никитину Алексею Александровичу

*фамилия, имя, отчество*

**Тема**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Утверждена приказом ректора от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_№ \_\_\_\_\_\_\_

Срок выполнения работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Задание принял к исполнению: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*подпись ФИО*

Барнаул 2014 г.

**1 Исходные данные**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2. Содержание разделов работы**

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование разделов  работы и их содержание | Трудо-ёмкость,  % | Срок  выполнения | Консультант  (Ф.И.О., подпись) |
| 1 Расчетно-пояснительная записка |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 Графическая часть |  |  |  |
|  |  |  |  |
|  |  |  |  |

**3. Научно-библиографический поиск**

3.1. По научно-технической литературе просмотреть Реферативные журналы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ за последние \_\_\_\_ года и научно-технические журналы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ за последние \_\_\_\_ года.

3.2. По нормативной литературе просмотреть указатели государственных и отраслевых стандартов за последний год.

3.3. Патентный поиск провести за \_\_\_\_ лет по странам \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель работы: Старолетов С.М.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Ф.И.О. подпись*



**Руководство пользователя**



**Исходный код программы**

1. Объемы рынка игр

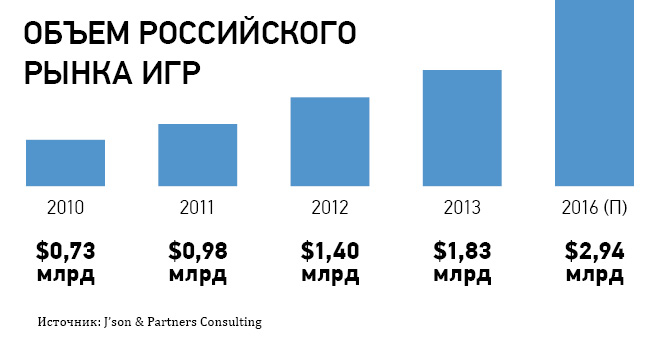
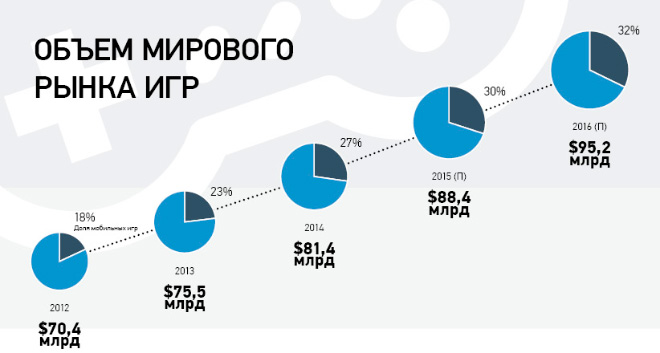
По оценке J’son & Partenrs Consulting, в 2015 году объем мирового рынка игр составит 88,4 миллиарда долларов, а объем российского рынка игр составит 1,83 миллиарда долларов. В целом рынок игр будет стабильно развиваться.

Рисунок Г.1 - Объем мирового рынка игр

Рисунок Г.2 - Объем российского рынка игр

1. Деревья поведения подробно описаны в главе 1. [↑](#footnote-ref-1)
2. Несущественные объекты – это игровые объекты, такие как массовка, животные, птицы, исключая ситуации, когда именно эти персонажи – главные герои в играх. Поведение несущественных объектов никак не влияет на ход игры. [↑](#footnote-ref-2)
3. В данном случае дерево поведения не должно содержать циклов даже если не учитывать направления ребер. [↑](#footnote-ref-3)