МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

**ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И
ТЕХНОЛОГИЧЕСКОГО ОБРАЗОВАНИЯ**
Кафедра информационных технологий и электронного обучения

Основная профессиональная образовательная программа
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль) «Технологии разработки программного обеспечения»
форма обучения – очная

Курсовая работа

«Управление программным проектом по разработке стратегической игры»

Обучающегося 3 курса
Афанасьева Андрея Дмитриевича

Научный руководитель:
Кандидат физико-математических наук,
доцент кафедры ИТиЭО
Жуков Николай Николаевич

Санкт-Петербург
2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Не секрет, что в наше время цифровые и компьютерные технологии обширно используются во всех сферах нашей жизни, и одной из них, является сфера развлечений. Различные игры, фильмы, видео, музыка и книги, все это сейчас можно легко и просто воспроизвести, используя компьютерные технологии. И наиболее прибыльной сферой, является разработка игр, как для персональных компьютеров, так и для телефонов и иных гаджетов. Связанно это с широким распространением технологий, а так же легкостью в их приобретения. Уже почти и не встретишь человека, у которого нет компьютера, или телефона. Такое обилие устройств и породило в каком-то смысле чудовищный спрос на игровую продукцию, что в свою очередь, подстегнуло как отдельных личностей, так и целые кампании всерьез заняться разработкой игр.

Процесс создания, или же разработки игры, крайне интересный и захватывающий процесс, и наряду с этим, сложный и затратный. Независимо от степени тяжести и объема проекта, разработка может предоставить разработчику множество проблем и сложностей, которые ему будет необходимо решить, что в свою очередь требует четкого контроля над разработкой продукта, и создания плана, которого будет необходимо придерживаться.

Со всем этим мне пришлось столкнуться при написании данной курсовой работы, так как её цель, создание игры в жанре «Tower Defense» (оборона башни), для персональных компьютеров. Данный жанр игр является одним из популярнейших, так как не требует “мощного железа” для работы, но при этом может дарить множество положительных эмоций. Кроме того, данный жанр игр хорошо развивает стратегическое мышление, и помогает учиться менеджменту ресурсов.

Для создания игры был использован специальный конструктор, под названием «Game Maker», за авторством нидерландского ученого Марка Овермаса. Версия, используемая при создании Game Maker 8.0 Pro.

Целью данной курсовой работы - является создание игры в жанре «Tower Defense», но с некоторыми отличиями от типичных представителей данного жанра.

Для достижения цели необходимо решить следующие задачи:

1)Выбрать среду программирования.

2)Создание плана, по которому будет разрабатываться игра.

3)Создание игры, и её последующее тестирование.

ТЕОРИТИЧЕСКАЯ ЧАСТЬ

1.1 Анализ agile-методологий (гибкая методология разработки) и выбор наиболее подходящей первый шаг при начале работы над проектом.

Гибкая методология разработки (англ. *agile software development, agile-разработка*) — обобщающий термин для целого ряда подходов и практик, основанных на ценностях Манифеста гибкой разработки программного обеспечения и 12 принципах, лежащих в его основе.

Большинство гибких методологий нацелены на минимизацию рисков путём сведения разработки к серии коротких циклов, называемых итерациями, которые обычно длятся две-три недели. Каждая итерация сама по себе выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требований, проектирование, программирование, тестирование и документирование.

На данный момент в IT-индустрии преобладают следующие методологии управления: SCRUM и Kanban.

Для выбора наиболее подходящего подхода, необходимо провести сравнительный анализ.

В описаниях этих методик много общего, и для наглядности анализа используем таблицу.

Scrum	Kanban
Продолжительность итераций/спринтов строго определена. Как правило, от двух недель до месяца.	Определяется именно продолжительность циклов.
Команда оценивает/планирует каждый спринт, опираясь на информацию из бэклога.	Отслеживается рабочий Kanban-карта
В этой методике участникам отводятся три роли: - владелец продукта, скрам-мастер, команда разработки.	Роли при процессе отсутствуют.
После начала спринта изменения не допускаются.	Изменения вносятся в любое удобное время.
Вся работа разбита на несколько спринтов.	Ход работы идет одним потоком.

На основе данной таблицы было принято решение воспользоваться методологией «SCRUM», так как она обеспечивает максимальную гибкость и контроль при разработке, что очень важно при разработке игры. Кроме того, происходит постоянный контакт команд, занимающихся разработкой.

1.2 Для реализации подхода SCRUM был использован онлайн-сервис Trello. Его аналоговый подход, с наклеиванием стикеров на виртуальную стену и записи на них задач, является крайне удобным, ввиду того что стена, отведенная под задачи, остается чистой, так же есть доступ у сотрудников, работающих удаленно и исключена потеря этих стикеров с задачами.

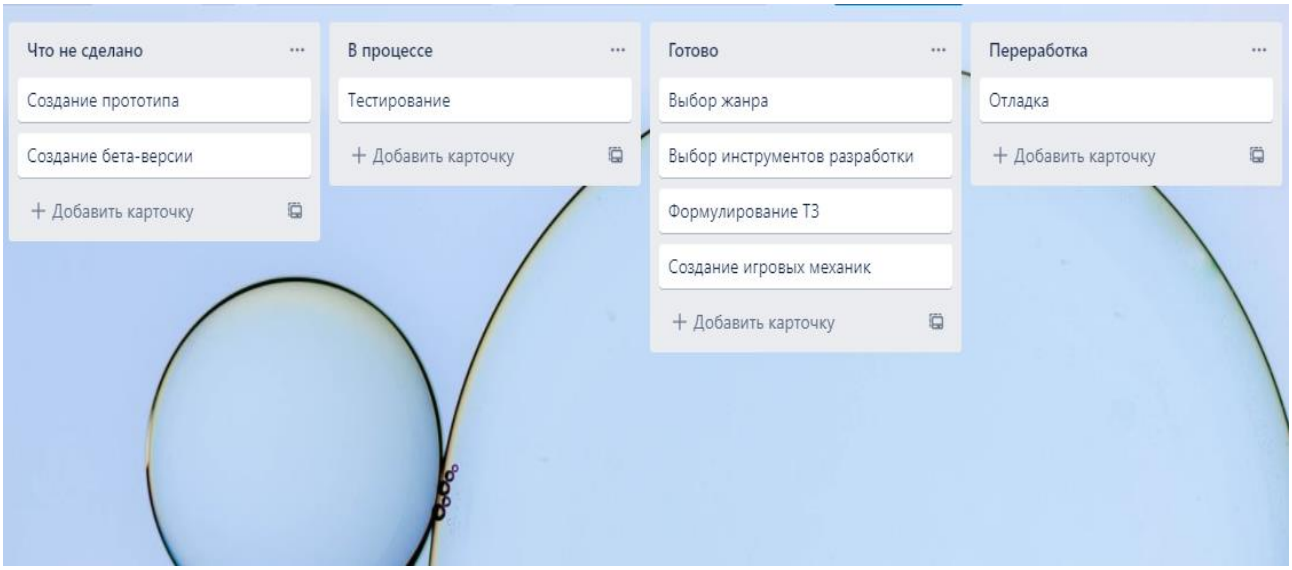
Trello, даже в базовой версии, предоставляет широкий спектр возможностей для контроля проектов: создание досок показывающих состояние проекта и состоящих из колонок, разделенных по определенному признаку, с задачами, а также возможность маркировать задания различными цветами. Для упрощения процесса работы над проектом, было создано 4 колонки (рис. 1):

- Что не сделано: Те этапы работа над которыми не начиналась, и ими необходимо заняться.

- В процессе: Этапы над которыми ведется работа прямо сейчас.

- Готово: Этапы работа над которыми завершена, и возвращаться к ним нет необходимости.

- Переработка: Этапы, работа над которыми началась по новой, ввиду необходимости начать их сначала.



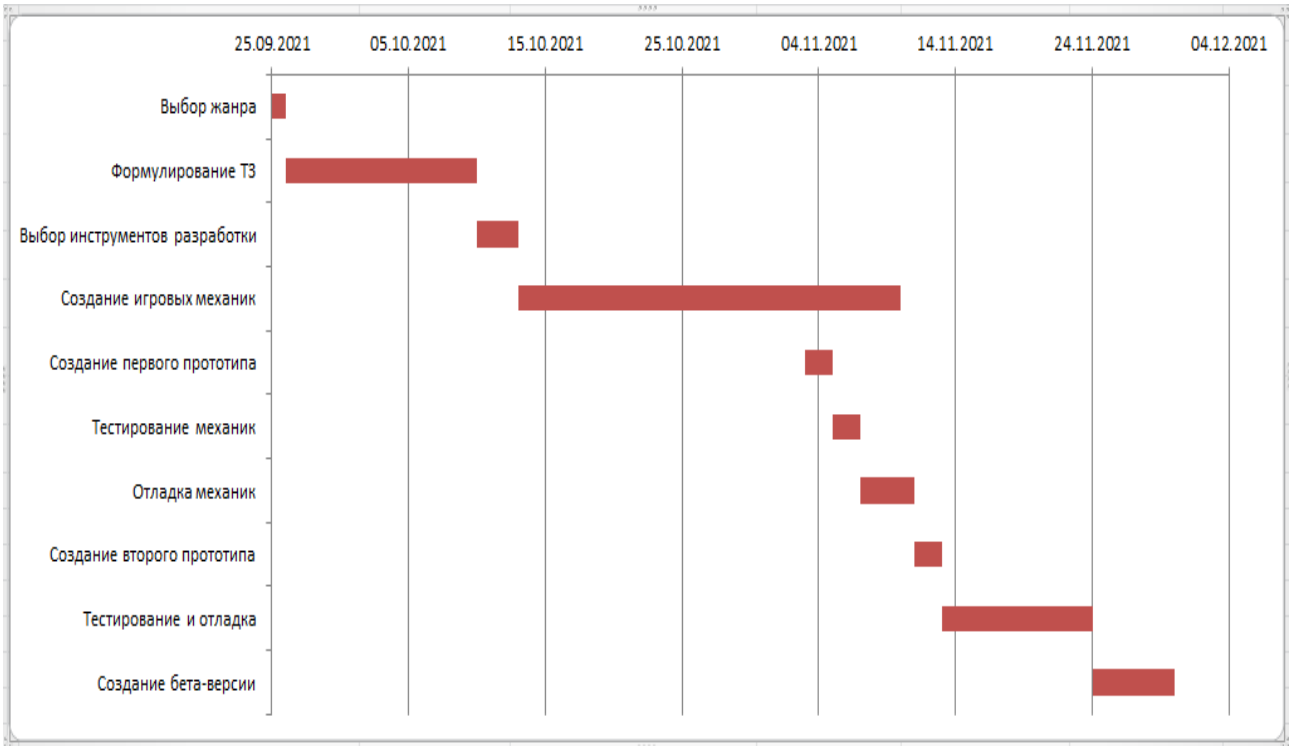
(рис. 1)

1.3 Для визуализации приблизительного плана разработки проекта, воспользуемся диаграммой Ганта (рис. 3).

Диаграмма Ганта — это популярный тип столбчатых диаграмм, который используется для иллюстрации плана, графика работ по проекту. Является одним из методов планирования проектов. Для построения диаграммы, будем использовать программу Microsoft Excel. Для начала создадим таблицу, которая будет представлять собой примерный план работы (рис. 2).

Название этапа	Начало	Предположительная длительность	Задержка	Длительность	Конец
Выбор жанра	25.09.2021	1	0	1	26.09.2021
Формулирование ТЗ	26.09.2021	14	0	14	10.10.2021
Выбор инструментов разработки	10.10.2021	3	0	3	13.10.2021
Создание игровых механик	13.10.2021	24	4	28	03.11.2021
Создание первого прототипа	03.11.2021	2	0	2	05.11.2021
Тестирование механик	05.11.2021	2	0	2	07.11.2021
Отладка механик	07.11.2021	3	1	4	11.11.2021
Создание второго прототипа	11.11.2021	2	0	2	13.11.2021
Тестирование и отладка	13.11.2021	10	1	11	24.11.2021
Создание бета-версии	24.11.2021	5	1	6	30.11.2021

(рис. 2)



(рис. 3)

1.4 Приложение «Ultimate Tower Defence» является комплексным проектом, охватывающим различные аспекты разработки ПО:

-содержит все основные аспекты компьютерной стратегической игры;

-содержит все основные аспекты компьютерной экономической игры;

- является продуктом сферы компьютерных развлечений.

Данный проект является компьютерной игрой, и из этого следует, что будет одна категория пользователей - игроки. В процессе работы приложения пользователь является непосредственным участником игрового процесса и оказывает влияние на него. Программа должна обладать следующим функционалом:

1) Графический функционал:

- Отрисовка элементов геймплея. (Враги, башня, карта)
- Отрисовка интерфейса
- Отрисовка huda

2) Звуковой функционал:

- Наличие звуковых эффектов основных элементов геймплея
- Наличие фоновой музыки

3) Внутри игровой функционал:

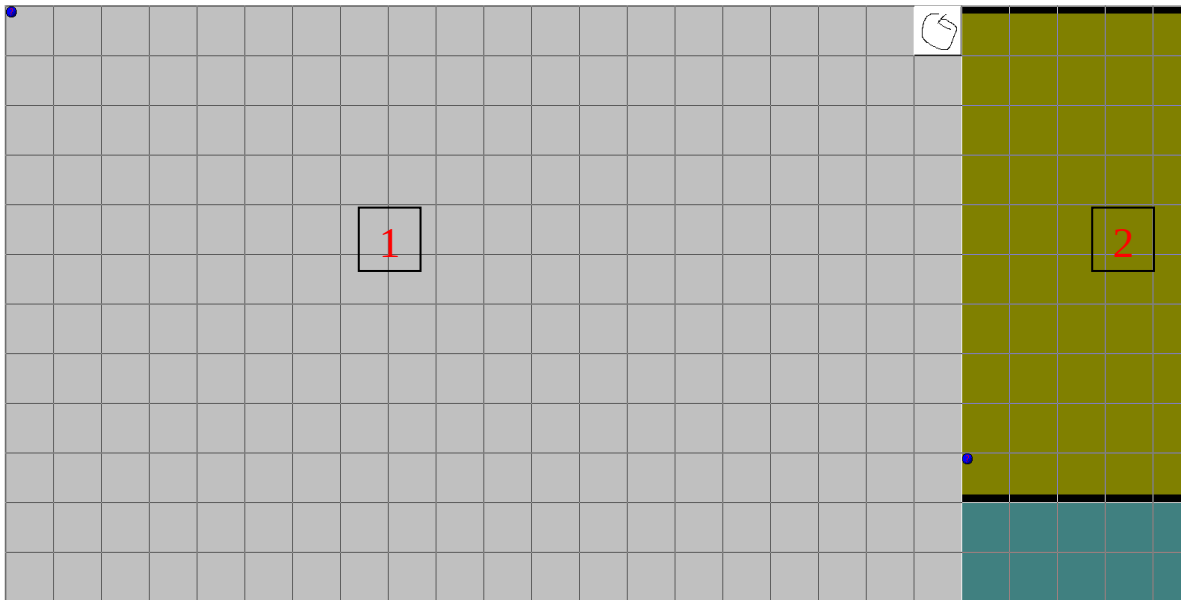
- Система случайной генерации уровня
- Система волнового появления противника
- Система покупки и улучшения башен

4)Пользовательский интерфейс:

- Показатель здоровья базы
- Показатель количества денег у игрока

1.5 Целью макета является визуальное и схематическое изображение внешнего вида приложения. Он необходим в первую очередь для дизайн-документа и создается игровым дизайнером совместно с дизайнером уровней. Макет позволяет наглядно увидеть интерфейс будущей программы, и на основе обратной связи, полученной от заказчика, сделать правки. Макет четко показывает топологию графических объектов в окне приложения.

Для данной игры необходим один макет, ввиду наличия в игре одного окна (рис. 4).



(рис. 4)

Игровой экран будет делиться на две зоны, первая зона, (1) является зоной уровня, где будет отображаться карта, противники и башни. Вторая зона (2),

предназначена для отрисовки счетчиков здоровья и денег, а так же для магазина и зоны улучшений.

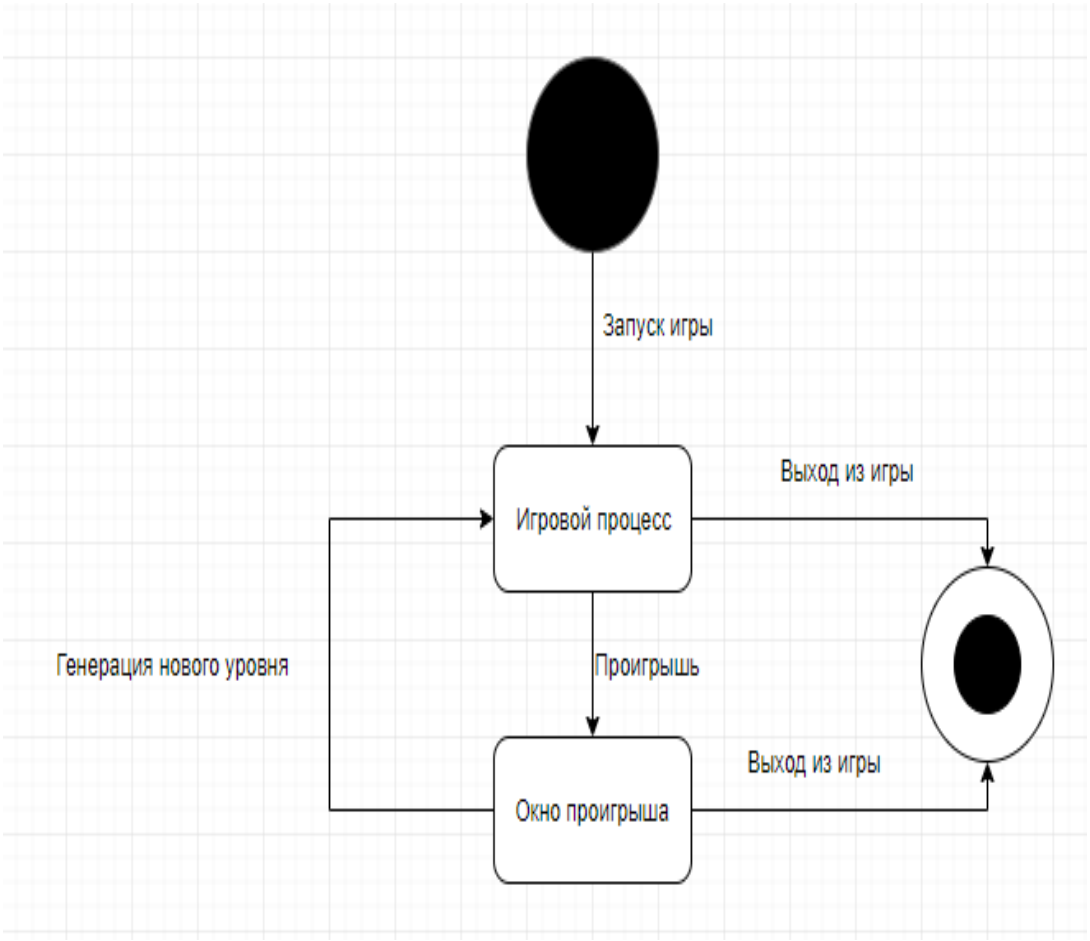
1.6 **Диаграмма состояний** — это, по существу, диаграмма состояний из теории автоматов со стандартизированными условными обозначениями, которая может определять множество систем от компьютерных программ до бизнес-процессов. Используются следующие условные обозначения:

- Круг, обозначающий начальное состояние.

- Окружность с маленьким кругом внутри, обозначающая конечное состояние (если есть).

- Скруглённый прямоугольник, обозначающий состояние. Верхушка прямоугольника содержит название состояния. В середине может быть горизонтальная линия, под которой записываются активности, происходящие в данном состоянии.

- Стрелка, обозначающая переход. Название события (если есть), вызывающего переход, отмечается рядом со стрелкой (рис. 5).


Запуск игры
Игровой процесс
Выход из игры
Генерация нового уровня
Проигрышь
Окно проигрыша
Выход из игры

(рис. 5)

ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Первое, что необходимо сделать, это написать программу для создания карты уровня, причем необходимо как соблюсти некоторые условия, а именно наличие дороги от места появления врагов до базы, наличие мест под башни, отсутствие тупиковых ответвлений и доступ к базе лишь по одному направлению, но при этом необходимо и следовать плану, то есть сделать так, чтобы при каждой генерации, карта была новой.

Сделать это можно опираясь на систему комнат в Game Maker. Каждая комната, это та или иная часть игры. Комната может являться как меню, так и быть уровнем, все зависит от наполнения. Каждая комната построена по принципу координатной сетки, где верхний правый угол имеет нулевую координату. При этом отрицательные координаты отсутствуют. Спускаясь по оси Y, мы будем получать положительные числа. То есть, спустившись на 10 условных единиц по Y, мы получим координаты X равного 0, и Y равного 10.

При этом, мы можем задать размер комнаты, а так же размер клеток, на которые эта комната будет разбита. Мы возьмем размер клеток 64x64, а размер комнаты 1600x768. Сделано это для того, что бы комнаты разбилась на цельные клетки, и что бы каждые координаты кратные 64 были пустой клеткой.

Теперь зная все это, мы можем представить каждую клетку в комнате, как часть двумерного массива, где в качестве номера элемента, выступают координаты. То есть, если номер массива у нас 128 и 64, это значит, что этот элемент отвечает за клетку, находящуюся на координатах X равный 128, и Y равный 64.

Это позволит нам работать с массивом, который после преобразуется в наш уровень. И что бы сделать это, необходимо добавить числа, которые будут означать тот или иной вид клетки. Пустое место, на котором ничего нет, будет обозначаться 0. Этому значению равны все элементы изначально. При 1, элемент будет превращен в дорогу, по которой будут идти противники. 2 означает стену, которая блокирует возможность уйти в сторону, а 3, место под

будущую башню. База и место для появления противника получаю значения 5 и 4 соответственно.

Для начала генерации уровня, необходимо выбрать случайную точку, где у нас появится база, от которой мы будем строить дорогу. Так же стоит учитывать, что точку брать надо не на краю комнаты, так как это может негативно сказать на дальнейшей генерации (рис. 6).

```
1 c = irandom_range(128,1152)
2 c = round(c/64) * 64
3 m = irandom_range(128,640)
4 m = round(m/64) * 64
5 instance_create(c,m,obj_base)
6 obj_contr.x=obj_base.x
7 obj_contr.y=obj_base.y
8 scr_create()
9
```

(рис. 6)

Первые четыре строчки выбирают случайные значения для X и Y, после чего эти значения приводят к ближайшему кратному 64, так как необходимо установить базу ровно в одну клетку. Если хоть одно из значений будет не кратно 64, то база появится на две, или более клеток, что отразиться на дальнейшей генерации негативно. После получения координат, мы на них создаем базу, после чего на этих же координатах создаем объект контроля, необходимый для дальнейшей работы. И в финале, мы запускам скрипт create.

Данный скрипт и дает начало остальной генерации уровня. Суть скрипта достаточно проста. Он изначально и создает вышеописанный массив, и тут же дает одному из элементов значение 5, так как этот элемент уже занят базой, и не может получить другое значение (рис. 7).

```
for (i=0;i<=1280;i+=64)          //создание массива координат
{
    for (j=0;j<=768;j+=64)
    {
        m[i, j] = 0
        global.road[i, j] = 0
    }
}
for (i=0;i<=1280;i+=1)
{
    for (j=0;j<=768;j+=1)
    {

        global.road[i, j] = 0
    }
}
m[obj_contr.x, obj_contr.y] = 5
```

(рис. 7)

Далее скрипт заглядывает через клетку, дабы проверить, упрется он в край комнаты или нет. Если такое происходит, то это направление блокируется, давая элементу с координатами соседней клетки значение 2 (рис. 8).

```
{
    r = irandom(100)
    if (obj_contr.x+128>1216)
    {
       m[obj_contr.x+64, obj_contr.y] = 2                    //если через клетку край комнаты, поставить на соседней клетке стену
    }
    if (obj_contr.x-128<0)
    {
       m[obj_contr.x-64, obj_contr.y] = 2                   //если через клетку край комнаты, поставить на соседней клетке стену
    }
    if  (obj_contr.y+128>704)
    {
       m[obj_contr.x,obj_contr.y+64] = 2                     //если через клетку край комнаты, поставить на соседней клетке стену
    }
    if (obj_contr.y-128<0)
    {
       m[obj_contr.x,obj_contr.y-64] = 2                     //если через клетку край комнаты, поставить на соседней клетке стену
    }
```

рис.8

Если же все нормально, и соседние клетки не заняты, то есть элементы массива с их координатами равны 0, скрипт случайно генерирует число, и если оно оказывается больше 3, то работа продолжается, и число снова генерируется. Далее в зависимости от числа, происходит создание дороги, по направлению вверху, снизу, либо справа или слева. Кроме того, происходит проверка на то, не будет ли столкновения с другой клеткой дороги, так как если это произойдет, дорога не сможет продолжить генерацию, так как дорога за кольцуется. Если же все проверки пройдены успешно, то элемент массива принимает значение в 1, а объект контроля сдвигается на эту координату, после чего повторяет вышеуказанные действия (рис. 9)

```
r = irandom(100)
if  (r <= 25) && (m[obj_contr.x+64, obj_contr.y] == 0)          //построить дорогу
    {
    if (m[obj_contr.x+64, obj_contr.y+64] == 0) && (m[obj_contr.x+64, obj_contr.y-64] == 0)     //проверка на соседство с дорогой. Если через клетку дорога, то строить нельзя.
    {
    if ((m[obj_contr.x+128, obj_contr.y] =0 ) || (m[obj_contr.x+128, obj_contr.y] =2 ))
    {
        m[obj_contr.x+64, obj_contr.y] = 1
        global.road[obj_contr.x+64, obj_contr.y] = 1
        obj_contr.x = obj_contr.x+64
    }
    }
    }

if (r > 25) && (r <= 50) && (m[obj_contr.x-64, obj_contr.y] == 0)   //построить дорогу
    {
    if ((m[obj_contr.x-128, obj_contr.y] = 0) || (m[obj_contr.x-128, obj_contr.y] = 2))     //проверка на соседство с дорогой. Если через клетку дорога, то строить нельзя.
    {
    if (m[obj_contr.x-64, obj_contr.y+64] == 0) && m[obj_contr.x-64, obj_contr.y-64] == 0
    {
        m[obj_contr.x-64, obj_contr.y] = 1
        global.road[obj_contr.x-64, obj_contr.y] = 1
        obj_contr.x = obj_contr.x-64
    }
    }
    }


if (r > 50) && (r <= 75) && (m[obj_contr.x, obj_contr.y-64] == 0)          //построить дорогу
    {
    if ((m[obj_contr.x, obj_contr.y-128] = 0) || (m[obj_contr.x, obj_contr.y-128] = 2))     //проверка на соседство с дорогой. Если через клетку дорога, то строить нельзя.
    {
    if (m[obj_contr.x+64, obj_contr.y-64] == 0) && (m[obj_contr.x-64, obj_contr.y-64] == 0)
    {
        m[obj_contr.x, obj_contr.y-64] = 1
        global.road[obj_contr.x, obj_contr.y-64] = 1
        obj_contr.y = obj_contr.y-64
    }
    }
    }

if (r > 75) && (r <= 100) && (m[obj_contr.x, obj_contr.y+64] == 0)  //построить дорогу
    {
    if ((m[obj_contr.x, obj_contr.y+128] = 0) || (m[obj_contr.x, obj_contr.y+128] = 2))     //проверка на соседство с дорогой. Если через клетку дорога, то строить нельзя.
    {
        if (m[obj_contr.x-64, obj_contr.y+64] == 0) && (m[obj_contr.x+64, obj_contr.y+64] == 0)
        {
        m[obj_contr.x, obj_contr.y+64] = 1
        global.road[obj_contr.x, obj_contr.y+64] = 1
        obj_contr.y = obj_contr.y+64
        }
    }
```

рис. 9

В случае если при первой генерации числа, число выпадает равное 3, или меньше, то мы заканчиваем создание дороги, и ищем место, где будет расположено место появления противника (рис. 10)

```
while (p = 1)
{
    for (i=0;i<=1280;i+=64)
    {
        for (j=0;j<=768;j+=64)
        {
            if (m[i, j] = 1)
            {
                rc = 0
                if (m[i+64, j] = 1) || (m[i+64, j] = 5) || (m[i+64, j] = 4)
                {
                    rc = rc + 1
                }
                if (m[i-64, j] = 1) || (m[i-64, j] = 5) || (m[i-64, j] = 4)
                {
                    rc = rc + 1
                }
                if (m[i, j+64] = 1) || (m[i, j+64] = 5) || (m[i, j+64] = 4)
                {
                    rc = rc + 1
                }
                if (m[i, j-64] = 1) || (m[i, j-64] = 5) || (m[i, j-64] = 4)
                {
                    rc = rc + 1
                }
                if rc = 1
                {
                    m[i, j] = 4
                    p = 2
                }
                rc = 0
            }

        }
    }
}
```

рис. 10

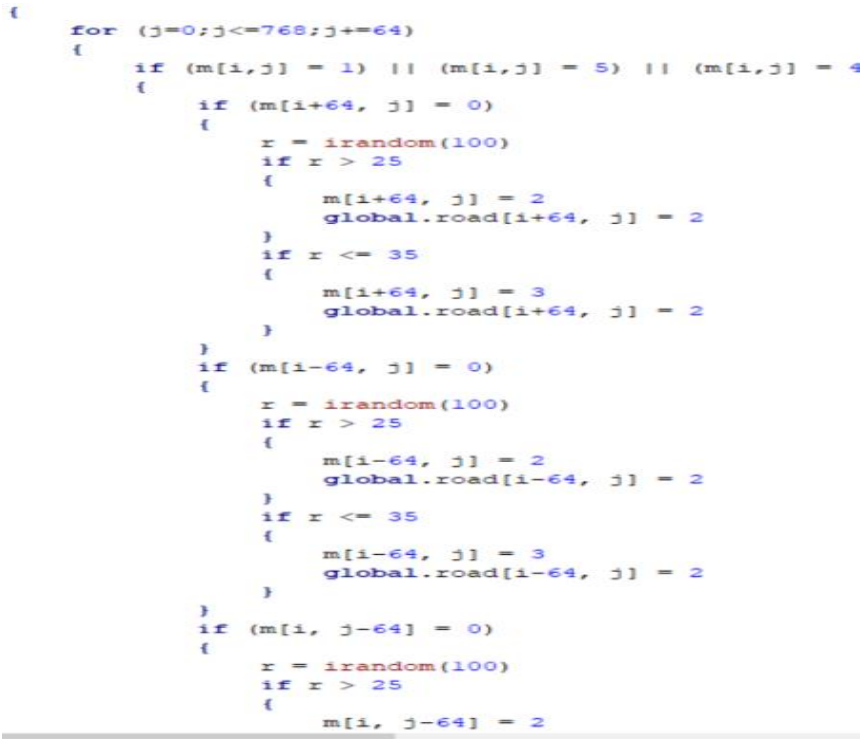
Подбор места происходит следующим образом: мы перебираем все элементы массива, дабы найти тот, который имеет лишь один соседний элемент со значением 1, то есть, ищем ту клетку, куда идет только одна дорога. Сделано это за тем, что бы у врага был лишь один возможный путь к базе.

В случае, если мы запускаем скрипт, и при первой же генерации числа необходимо установить место для появления противника, то есть выходит так что база и враг будут вплотную друг к другу, то мы случайным образом выбираем, где будет поставлена позиция противников, сверху, снизу, справа или слева (рис. 11).

```
{
    if m[i,j] = 5
    {
        if (m[i+64, j] <> 1) && (m[i-64, j] <> 1) && (m[i, j+64] <> 1) && (m[i, j-64] <> 1)
        {
            d = 0
            while (d = 0)
            {
            r = irandom(100)
            if (r <= 25) && (m[i+64, j] <> 2)
            {
                m[i+64, j] = 4
                d = 1
                p = 2
            }
            if (r > 25) && (r <= 50) && (m[i-64, j] <> 2)
            {
                m[i-64, j] = 4
                d = 1
                p = 2
            }
            if (r > 50) && (r <= 75) && (m[i, j+64] <> 2)
            {
                m[i, j+64] = 4
                d = 1
                p = 2
            }
            if (r > 75) && (m[i, j-64] <> 2)
            {
                m[i, j-64] = 4
                d = 1
                p = 2
            }
            }
        }
    }
}
```

рис. 11

После того как база, место под появление противника, и дорога были установлены, мы генерируем окружающие их стены, и места под создание башен (рис. 12).



```
{
    for (j=0;j<=768;j+=64)
    {
        if (m[i,j] = 1) || (m[i,j] = 5) || (m[i,j] = 4
        {
            if (m[i+64, j] = 0)
            {
                r = irandom(100)
                if r > 25
                {
                    m[i+64, j] = 2
                    global.road[i+64, j] = 2
                }
                if r <= 35
                {
                    m[i+64, j] = 3
                    global.road[i+64, j] = 2
                }
            }
            if (m[i-64, j] = 0)
            {
                r = irandom(100)
                if r > 25
                {
                    m[i-64, j] = 2
                    global.road[i-64, j] = 2
                }
                if r <= 35
                {
                    m[i-64, j] = 3
                    global.road[i-64, j] = 2
                }
            }
            if (m[i, j-64] = 0)
            {
                r = irandom(100)
                if r > 25
                {
                    m[i, j-64] = 2
```

рис. 12

После того как мы закончили генерацию стен, наступает финальная часть скрипта. Имея двумерный массив с числовой картой нашего уровня, мы начинаем переносить её в комнату. Используя номер элемента, мы вычисляем его координаты, и в зависимости от его числового значения, генерируем тот или иной объект в комнате (рис. 13).

```
for (i=0;i<=1280;i+=64)
{
    for (j=0;j<=768;j+=64)
    {
        if m[i,j] = 1
        {
            instance_create(i,j,obj_road)
        }
        if m[i,j] = 2
        {
            instance_create(i,j,obj_block)
        }
        if m[i,j] = 3
        {
            instance_create(i,j,obj_place)
        }
        if m[i,j] = 4
        {
            instance_create(i,j,obj_portal)
        }
    }

}
```

рис. 13

После этого, игра показывает сгенерированный уровень, который при каждой новой генерации должен иметь новый вид. Можем это протестировать, дабы убедиться в работе скрипта, или наоборот, найти ошибки и баги (рис. 14-16).

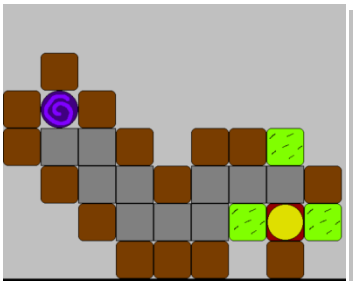


рис. 14

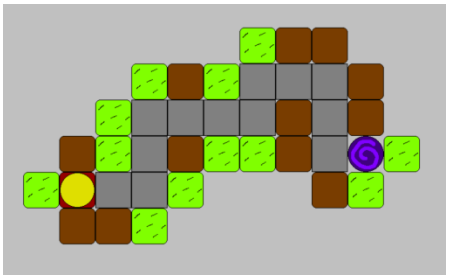


рис. 15

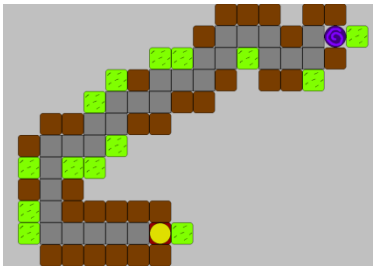


рис. 16

Как можно увидеть, скрипт работает, как мы и планировали. Каждая генерация, дарит нам уникальный уровень, не схожий с предыдущими. Первая часть плана выполнена.

2.2 После генерации уровня, необходимо озаботиться теми, кто и бросает вызов игру, а именно противники. Следуя плану, нам необходимо сделать их появление волнами, со случайным числом врагов в каждой волне, а так же, что бы каждый противник имел свои, случайные характеристики.

Для создания это системы, необходим новый объект, который будет запускать волны. Сделаем его в виде кнопки, и настроим (рис. 17).

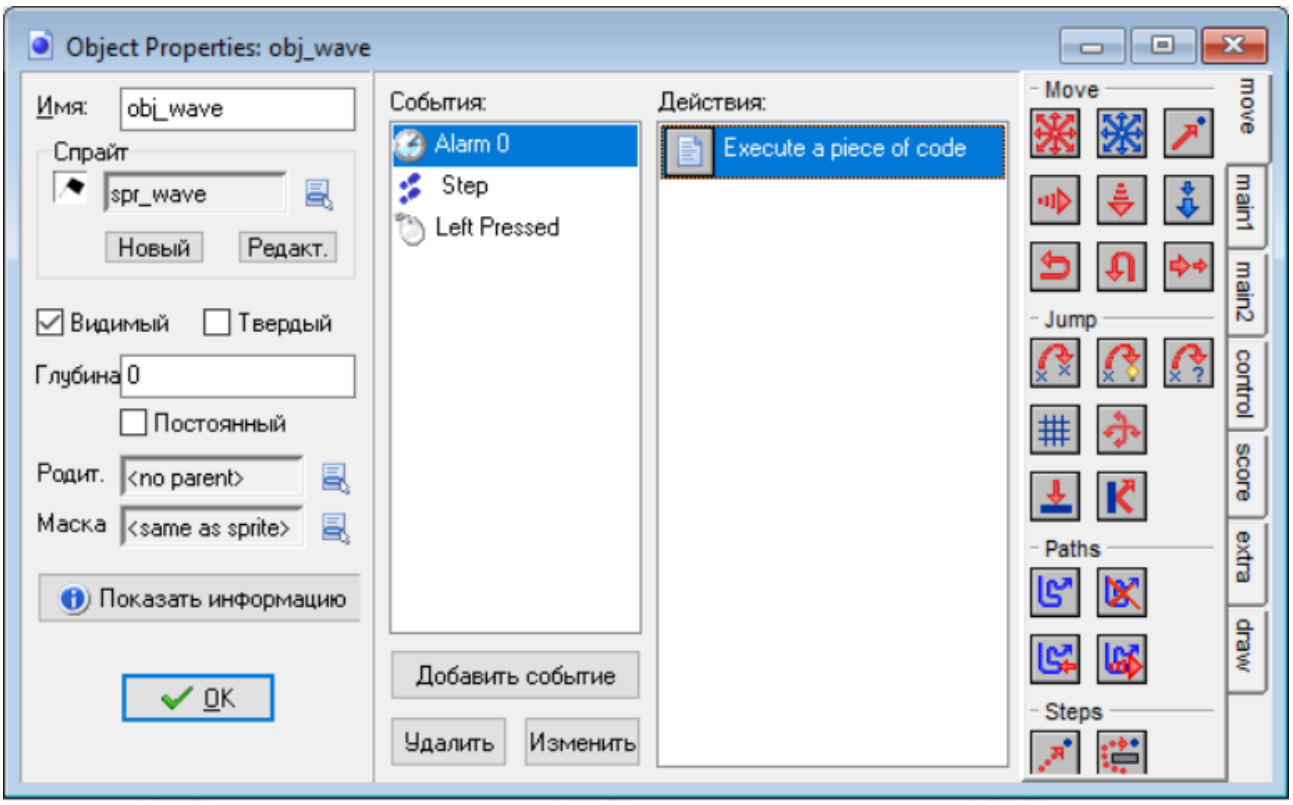


рис. 17

Рассмотрим события в порядке их важности. Первым идет событие step. Данное событие работает как таймер, который каждые несколько тиков, запускает действия, указанные в нем. В нашем случае, в step находится код, проверяющий, идет сейчас волна или нет (рис. 18).

```
if global.wave_now = 1
{
    if a = 0
    {
        alarm[0] = 20
        a = 1
    }
    if spawn = r
    {
        global.wave_now = 0
    }
}
```

рис. 18

Если в данный момент идет волна, объект каждые 20 тиков, обращается к таймеру, который можно так же увидеть среди событий. Таймер в свою очередь, создает противника, и контролирует количество врагов на волну (рис. 19).

```
instance_create(obj_portal.x,obj_portal.y,obj_gob)
spawn = spawn + 1
a = 0
```

рис. 19

Сама волна запускается по нажатию мышкой на объект. Этим же действием определяется число противников, которые появятся на этой волне, а так же повышает сложность игры. Каждые 5 волн, враги получат слабое усиление, а каждые 10 большое (рис. 20).

```
if global.wave_now = 0
{
    a = 0
    spawn = 0
    r = irandom_range(global.wave, global.wave*3)
    global.wave_now = 1
    global.wave = global.wave + 1
    if global.wave mod 10 = 0
    {
        global.minhp = global.minhp + 15
        global.maxhp = global.maxhp + 30
        global.mindef = global.mindef + 2
        global.maxdef = global.maxdef + 2
    }
    if global.wave mod 5 = 0
    {
        global.minhp = global.minhp + 10
        global.maxhp = global.maxhp + 20
        global.mindef = global.mindef + 1
        global.maxdef = global.maxdef + 1
    }
}
```

рис. 20

Теперь, когда система создания противников готова, переходим к самим противникам. Так как каждый враг у нас уникален, дадим объекту врага несколько спрайтов, Один из которых он и будет случайно брать при появлении. Кроме того, при появлении враг так же случайным образом будет определять свои характеристики, выбирая их между минимальным и максимальным значением. Самих характеристик у нас будет четыре. Количество жизней, скорость, защита, и шанс на срабатывание этой самой защиты. Как уж было указано, каждая из характеристик будет формироваться

между двумя значениями, которые с увеличением номера волны, будут так же расти. То есть, на первой волне, минимально возможное значение здоровья противника 15, а максимальное 40, то уже на волне 10, эти значения будут 25 и 50 соответственно. Благодаря этому, противник, как и игрок, не будет стоять на месте, а будет постепенно развиваться, становясь все сильнее и сильнее, тем самым подогревая интерес к игре (рис. 21).

```
self.hp = irandom_range(global.minhp,global.maxhp)
self.s = irandom_range(global.minspeed,global.maxspeed)
self.def = irandom_range(global.mindef,global.maxdef)
self.chance_of_def = irandom_range(global.min_chance_of_def,global.max_chance_of_def)
self.collision = 0
self.sprite_index=choose(spr_gob,spr_gob1,spr_gob2,spr_gob3)
self.exp_dmg = 1
self.on_fire = 0
```

рис. 21

Последнее что необходимо сделать с противника в данный момент, научить их передвигаться. И с этим, есть проблемы. Как была указано ранее, в Game Maker используются координаты, и когда какой-либо из объектов начинает передвигаться, он передвигается на n-ное число координат. Это означает, что если у объекта скорость 10, то его координаты будут менять на десять сразу, а не постепенно, что в обычных условиях не критично, но в случае с нашей игрой, очень большая проблема. Так как каждый раз мы получаем новую карту, создать заранее прописанный маршрут нельзя. Стандартные функции поиска пути, встроенные в редактор, так же не помощники нам, ввиду все той же случайно генерации. При попытке использовать встроенные функции, зачатую враг рано или поздно застревает на поворотах, или ещё раньше, после чего не в силах выбраться, и продолжить путь. Решение проблемы есть, но не самое удачное.

Вернемся к скрипту create, и добавив к нему новую функцию. Так как после завершения строительства дороги, объект контроля никуда не исчез, воспользуемся им вновь. На всякий случай переместим его в координаты

портала, и начнем движение к базе, как бы имитируя противника. Находя ближайшую дорогу, объект контроля перемещается к ней, оставляя на прошлой клетке указание, куда двигаться. Оказавшись на дороге, контролер ищет новую клетку, где не был до этого, после чего вновь перемещается к ней, оставляя очередное указание пути. Далее, повторяя этот алгоритм, контролер добирается до базы, где останавливается, и заканчивает работу, оставляя позади себя дорогу из указаний, как идти. По итогу, противник когда появляется, идет не совсем к базе, а следует от указателя к указателю, которые и выводят его к базе (рис. 22-23).

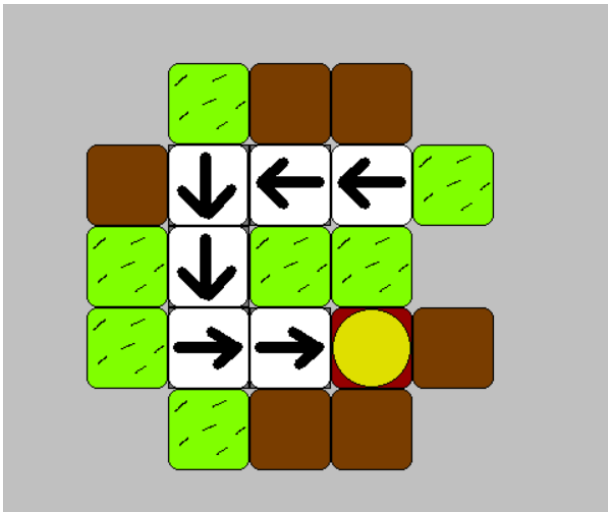


рис. 22

```
obj_contr.x = obj_portal.x
obj_contr.y = obj_portal.y
lastx = 0
lasty = 0
b = 0
while (b = 0)
{
    if (m[obj_contr.x+64,obj_contr.y] = 1) && (lastx <> obj_contr.x+64)
    {
        instance_create(obj_contr.x,obj_contr.y,obj_rightturn)
        lastx = obj_contr.x
        lasty = 0
        obj_contr.x = obj_contr.x+64
    }
    if (m[obj_contr.x-64,obj_contr.y] = 1) && (lastx <> obj_contr.x-64)
    {
        instance_create(obj_contr.x,obj_contr.y,obj_leftturn)
        lastx = obj_contr.x
        lasty = 0
        obj_contr.x = obj_contr.x-64
    }
    if (m[obj_contr.x,obj_contr.y+64] = 1) && (lasty <> obj_contr.y+64)
    {
        instance_create(obj_contr.x,obj_contr.y,obj_downturn)
        lasty = obj_contr.y
        lastx = 0
        obj_contr.y = obj_contr.y+64
    }
    if (m[obj_contr.x,obj_contr.y-64] = 1) && (lasty <> obj_contr.y-64)
    {
        instance_create(obj_contr.x,obj_contr.y,obj_upturn)
        lasty = obj_contr.y
        lastx = 0
        obj_contr.y = obj_contr.y-64
    }
```

рис. 23

Для игрока, данные блоки не видимы, что впрочем, не сказывается на их функциональности. Применить подобных подход к поиску пути к самим противникам нельзя, так как объект контроля мгновенно меняет свои координаты на координаты следующей клетки дороги, в то время как противнику необходимо преодолеть это расстояние плавно, дабы игрок мог его увидеть, и уничтожить.

Но даже такая система не лишена багов. Иногда, из за криво высчитывающихся координат, противник как бы не замечает указателя, и продолжает движения опираясь на прошлый, от чего, закономерно, упирается в стену, или в башню. Решить эту проблему удалось уже отредактировав объект противника, добавив ему условие, что при столкновении со стеной, башней или только местом под неё, враг отскакивает от неё, на ближайшие координаты кратные 64, где он уже гарантированно замечает указатель, и продолжает следовать нужной дорогой (рис. 24).

```
self.speed = 0
if (x mod 64 <> 0)
{
x = round(x / 64) * 64
}
if (y mod 64 <> 0)
{
y = round(y / 64) * 64
}
```

рис. 24

Теперь, основные функции противников готовы, и необходимо протестировать их. Для этого, так же запустим игру, и на разных уровнях, запусти волну, и посмотрим на поведение противника (рис. 25-27).

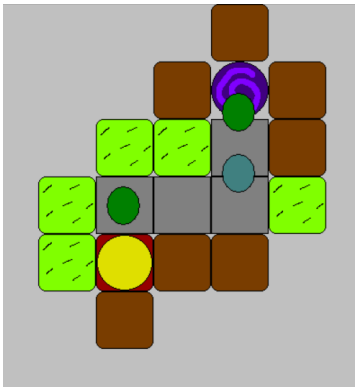


рис. 25

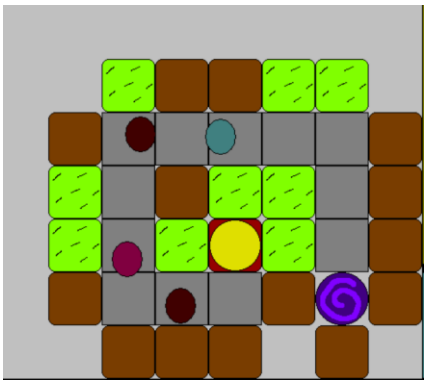


рис. 26

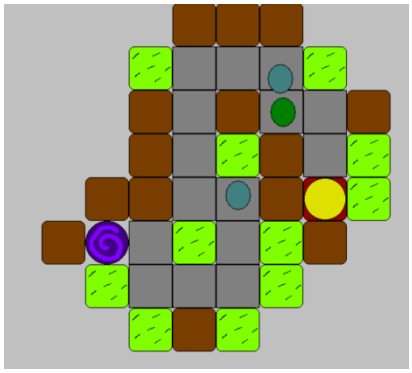


рис. 27

Проведя тестирование, проверяем работоспособность противников, и видим, что все работает. У врагов разный внешний вид, в пределах тех спрайтов что мы для них нарисовали, разные характеристики, что можно увидеть со скоростью, и так же враги добираются до базы, изредка врезаясь в стены, а значит, со второй частью плана мы справились.

2.3 Последнее что необходимо сделать, это возможность строить башни, и улучшать их. Для этого так же создать новый объект, а так же контуры меню покупки и улучшения (рис. 28).

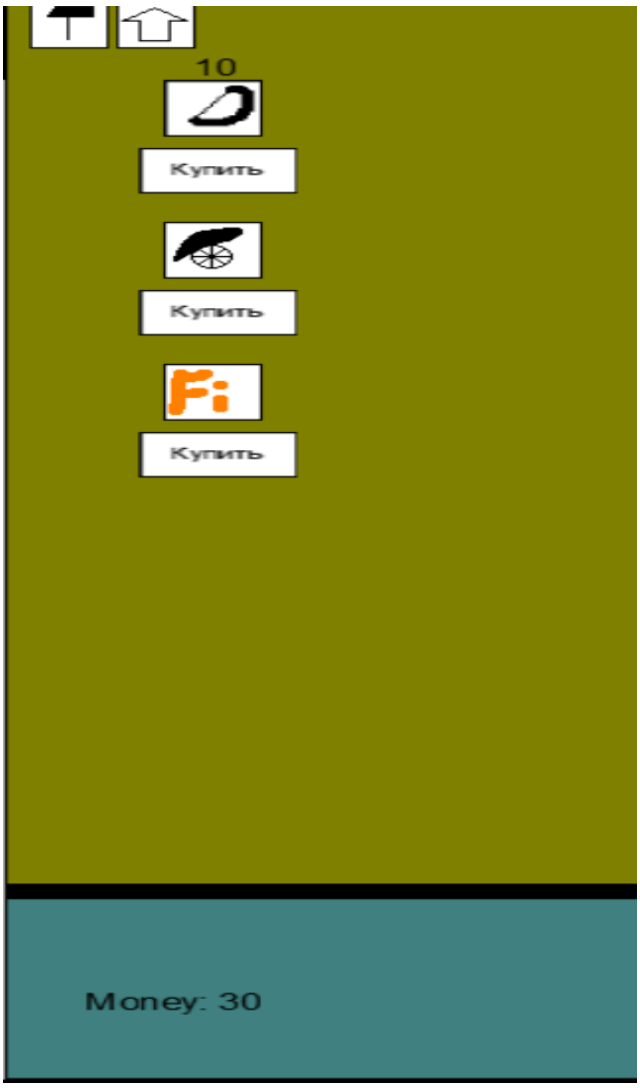


рис. 28

Пусть в меню у нас будет две вкладки. Одна под строительство, другая под улучшение. Переключение между ними будет посредством нажатия на соответствующую кнопку.

При нажатии на кнопку магазина, будет меняться значение переменных, на что среагирует объект контроля, и удалит объекты прошлого меню, и создаст те, которые нам необходимы в данный момент (рис. 29-31).

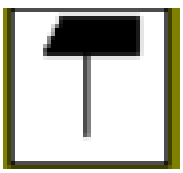


рис. 29

```
global.s = 1
global.u = 0
global.up_archer = 0
```

рис. 30

```
if global.s = 1 && instance_exists(obj_shop_arch) = 0 && instance_exists(obj_shop_can) = 0 && instance_exists(obj_shop_fire_wiz) = 0
{
    instance_create(1344,64,obj_shop_arch)
    instance_create(1344,164,obj_shop_can)
    instance_create(1344,264,obj_shop_fire_wiz)
}
```

рис. 31

К каждому типу башни добавим свою кнопку покупки, которая при нажатии будет менять наш курсор, а так же передавать значение переменной, для определения типа башни которую мы хотим купить и установить. (рис. 32).

```
global.ba = 1
if global.ba = 1 && global.money >= 10
{
    cursor_sprite = spr_curs_build
    global.money = global.money - 10
    global.trats = 10
}
```

рис. 32

Далее, если мы нажмем измененным курсором по площадке для создания башни, то та уничтожит себя, но перед этим, создаст на своих координатах выбранную нами башню (рис. 33-34).

```
if global.ba = 1
{
    instance_create(x,y,obj_archer)
    global.ba = 0
    cursor_sprite = spr_curs
    instance_destroy()
}


if global.bc = 1
{
    instance_create(x,y,obj_cannon)
    global.bc = 0
    cursor_sprite = spr_curs
    instance_destroy()
}


if global.bfw = 1
{
    instance_create(x,y,obj_fire_wiz)
    global.bfw = 0
    cursor_sprite = spr_curs
    instance_destroy()
}
```

рис. 33

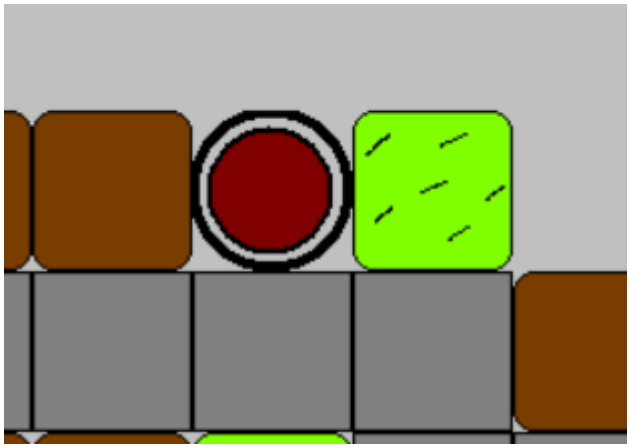


рис.34

Если же игрок передумал, и не хочет покупать данную башню, нажав правую кнопку мыши, покупка будет отменена, и затраченные средства вернуться на баланс игрока.

И вот, возможность строить башни добавлена, и работает. Теперь необходимо добавить возможность улучшать их. Повторяя прошлые действия, создаем объект для переключения на меню улучшений (рис. 35).

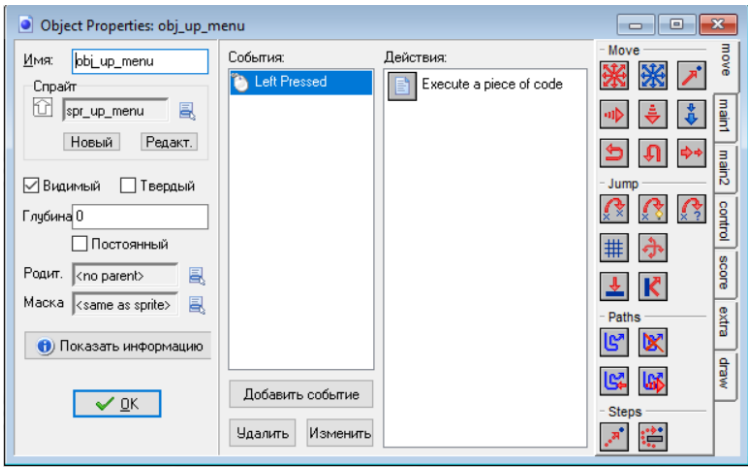


рис. 35

Аналогично с кнопкой меню строительства, эта при нажатии удаляет объекты старого меню, и создает новые. Используя их, мы и будем улучшать башни. Для этого, нажимаем на ярлык нужной башни, и открываем меню, где за указанную стоимость, мы можем провести улучшение (рис. 36).

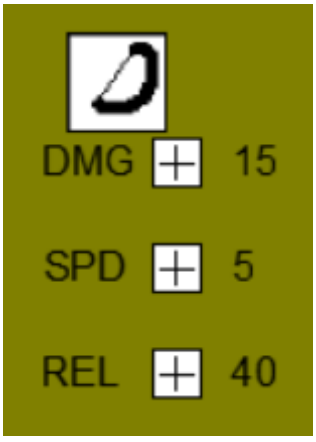


Рис. 36

После нажатия на кнопку, происходит повышение выбранного параметра на случайное число, а так же, увеличивается и цена на последующие улучшения (рис. 37)

```
if global.money >= global.arrow_damage_up
{
    global.arrow_damage = global.arrow_damage + irandom_range(1,10)
    global.money = global.money - global.arrow_damage_up
    global.arrow_damage_up = global.arrow_damage_up + irandom_range(global.arrow_up_min,global.arrow_up_max)
    r = irandom_range(1,5)
    if r = 1
    {
        global.minhp = global.minhp + irandom_range(5,35)
        global.maxhp = global.maxhp + irandom_range(10,60)
    }
    if r = 2
    {
        global.mindef = global.mindef + irandom_range(1,8)
        global.maxdef = global.maxdef + irandom_range(2,15)
    }
    if r = 3
    {
        global.min_chance_of_def = global.min_chance_of_def + irandom_range(0,5)
        global.max_chance_of_def = global.max_chance_of_def + irandom_range(5,10)
    }

}
```

Рис. 37

Так же, добавим шанс того, что при покупке улучшения, враг так же получит то или иное усиление. И вот теперь, наша система покупки-улучшения готова. Пора приступать к работе с самими башнями.

Так как башня должна стрелять только при приближении противника, задаем каждой из башен свой радиус. Как только враг заходит в радиус башни, она запускает таймер, который в зависимости от скорости стрельбы башни, создает снаряды, и продолжает это делать, до тех пор, пока враг находится в радиусе действия башни, или пока он жив (Рис. 38).

```
if (distance_to_object(obj_gob) < 256) && (self.e = 0)
{
    self.fire = 1
    self.e = 1
}
if (distance_to_object(obj_gob) > 256)
{
    self.fire = 0
    self.e = 0
}



if self.fire = 1
{
    instance_create(self.x+32,self.y+32,obj_arrow)
    sound_play(sou_arrow)
    alarm[0] = global.arch_reload
    self.fire = 0
}
```

рис. 38

При появлении, снаряд находит ближайшего противника, после чего направляется к нему, а точнее, к координатам, где он его нашел. Данная

особенность непреднамеренно добавляет в игру новую механику, а именно шанс промазать по цели, если скорость полета снаряда слишком маленькая (рис. 39).

```
self.target = instance_nearest(self.x,self.y,obj_gob)
image_angle = point_direction(self.x,self.y,self.target.x+32,self.target.y+32)
self.direction = point_direction(self.x,self.y,self.target.x+32,self.target.y+32)
self.speed = global.arrow_speed
```

рис. 39

Но с нахождение цели есть сложности. Так как каждый противник, это один и тот же объект, имеющий разный вид и характеристики, то если в качестве цели указывать именно объект, то снаряд будет двигаться за самым первым противником, не зависимо от того, находится он в радиусе действия башни или нет. Поэтому, для фокусировки на разных целях, снаряд обращается к id объекта. Происходит это так: появляясь, снаряд ищет вокруг себя ближайший id объекта, обозначенного в функции, а так как несколько копий одного объекта будут иметь свои, уникальные id, то и цель он сможет находить разную.

При попадании снаряда во врага, последний реагирует на это столкновение, и высчитывает, сколько урона он получает. Но этот расчет влияю как показатели снаряда, так и показатели самого противника. Так как у противника есть такой параметр как защита, то есть шанс того, что она сработает. Если врагу повезло, и его защита сработала, то вместо полного урона, он получит урон, минус показатель своей защиты, соответственно, если защита не сработала, то урон будет получен в чистом виде. А так как у каждого противника свой шанс на срабатывание защиты, и свой показатель защиты, то урон по каждому противнику проходит свой.

Кроме того, каждая башня имеет свои особенности. Например, башня лучников имеет наибольший начальный радиус, но при этом средний показатель урона, и скорострельности. Башня мага же, хоть и наносит мало урона, но поджигает противника, из-за чего, тот даже после попадания снаряд,

временно будет получать урон. Пушка же, имея наименьший радиус, и долгий перерыв между выстрелами, при попадании вызывает огненное облако, задевающее соседних противников.

Проведя тесты, можно убедиться, что система работает. Башни можно устанавливать на пригодные для этого позиции, те в свою очередь ведут огонь, каждая учитывая свои особенности.

ЗАКЛЮЧЕНИЕ

По итогу работы, можно сказать, что план выполнен, хотя назвать сделанную игру готовой нельзя. Ввиду отсутствия навыков, не вышло создать нормальное графическое оформление, оставив все на уровне тестовых фигур и кнопок. Присутствует необходимость переработки графической составляющей, а так же добавления звукового сопровождения. Кроме того, необходимо либо доделать, либо переделать с нуля систему перемещения противников, так как нынешняя имеет проблемы. Кроме того, имеется возможность расширения функционала игры, добавляя новые башни, а так же расширяя функционал существующих. Кроме того, можно добавить в игру сюжет, или оставить её соревновательной, добавив систему вычисления очков, и создав таблицу лидеров.

Можно сказать, что был сделан основной скелет игры, на который теперь можно добавить новые элементы, дабы придать игре законченный вид. Но учитывая некоторые моменты, становиться, очевидно, что силами одного разработчика создать полноценную игру крайне сложно, особенно не имея навыков работы с графикой и звуком. В ходе выполнения работы, были получены навыки разработки игры, в жанре Tower Defence, а так же теоретические навыки по разработке программ были применены на практике.

ЛИТЕРАТУРА

1. Scrum vs Kanban: в чем разница и что выбрать?

URL: https://habr.com/ru/company/hygger/blog/351048/

2. История жанра Tower Defense.

URL: https://hmong.ru/wiki/Tower_defense

3. Tower defense
URL: https://vlab.fandom.com/ru/wiki/Tower_defense

4.Справка по Game Maker.

URL: https://docs.yoyogames.com/source/dadiospice/002_reference/index.html

5. The Game Maker Language

URL: http://castle.pri.ee/wp-content/uploads/2010/10/Advanced-use_3.pdf