GAME MAKER ДЛЯ ЧАЙНИКОВ.

GML (GAME MAKER LANGUAGE)

GML – простой, Си или Паскале подобный, язык. Какой выберите это не важно, неважно для меня, но весь код будет в Си подобной манере. Итак, начинаем:

ЯЗЫК ПРОГРАММИРОВАНИЯ – интерфейс, который говорит компьютеру, что и как делать, так сказать язык общения с компьютером. Т.е. мы преобразуем человеческую логику в компьютерную, а язык программирования выполняет роль посредника.

```
f = 0
d=4;
s=3
if f = 0 if f==0 if(f=0) then s-=2;

if (f!=0) d+=2
```

КОД 1. ПРИМЕР ТОГО КАК НЕ НАДО ДЕЛАТЬ.

Игра – программа. Так? ☺

Отсюда следует вывод: Что все правила соблюдаемые при созданий программ, должны соблюдаться и для игр.

Должен быть отформатированный синтаксис, т.е. чистенький код. Это не для компьютера, но для удобства чтения. Компьютеру пофиг на форматирование текста. Посмотрите на Код 1. Видите? Я могу поставить «;» в конце строки, а могу и не поставить, могу оператор **if** заключить в скобки, а могу и не заключить, и т.д.

Вот эти вольности разрешены, но они путают программиста, особенно начинающего. Поэтому надо придерживаться определённых правил. Весёлые парадоксы, которые возникают (внезапно ☺), в коде я покажу чуть позже. Выберите свой стиль и придерживайтесь его на протяжений всего проекта. Самим будет удобнее.

1. Комментируйте свой код. Оставлять пометки. Чтобы не забыть, потом что есть что. Для "незабывчивых" – однажды забудете, я вам гарантирую, сам забывал. А потом возится с собственным кодом и думать: "А что ж я такое здесь делал то?"

В ГМ комментарий можно оставить 2 способами:

1. Комментирует строку после "//":

```
f  = 0;
f -= 2;    //вычитает.
f += 2;
```

2. Комментирует блок от "/*" до "*/":

```
f  = 0;
/*
Здесь
Будет
f -= 2;   <- не будет работать, т.к. эта строка входит в блок комментария
Моя Игра.
*/
f += 2;
```

Но знайте меру. Не надо комментировать совсем уж очевидные вещи:

```
if(d > m) //если d больше m.
```

Уже и так всё понятно. Лишние комментарий будут мешаться.

2. Давайте узнаваемые имена своим переменным.

Например, что лучше?
am = 30; или ammo = 30;
pra = 15; или primary_reload_ammo = 15;
Завтра вы догадаетесь что имели ввиду под именами "am" и "pra"? А послезавтра? А через неделю? Сокращения можно использовать, но если они постоянно «на слуху», повторяются ежедневно, например **var** (variable).

Вот два основных правила, которых вам придётся придерживаться, при кодинге. Остальные будут чуть позже.

Программа – это в основном идея, а не код. Сначала идея, потом код. "Зачем это говорить, и так всё понятно" – скажете вы, но ... На самом деле многие поступают наоборот. Сначала пишут код, а потом придумывают

идею. Если поступать так, то вскоре можно прийти в тупик. Программа так завяжется в узел, что исправить её будет невозможно. Проще её пристрелить (чтоб не мучилась) и переделать всё заново. Чтобы такого не произошло, делайте идею. В играх это выливается в разработку дизайн документа (диздок – короче). В нём хранятся все детали, все связи, все идеи будущего проекта. Диздок нужен в 2 случаях: Чтобы не забыть идеи и чтобы связать всех разработчиков одного проекта.

Теперь перейдём непосредственно к ГМ.

Основной синтаксис:

Программа

Блок программы выделяется фигурными скобками:

```
{
    //Здесь программа.
}
```

В фигурных скобках мы определяем, как бы, место действия. Если есть какой-то оператор он будет действовать в ограниченных вами рамках, в фигурных скобках.

Нам надо хранить где-то наши данные? Надо. Для этого используются…

Переменные:

Переменные хранят часть информаций в памяти. Например, число или текстовую строку. К переменной обращаемся по имени.

Пример:

```
ammo = 15; //здесь мы создали переменную ammo и присвоили значение 15.
k = 0;     //здесь мы создали переменную k и присвоили ей значение 0.
k = ammo;  /*здесь мы присвоили переменной k значение переменной ammo. И k теперь 15.*/
```

Логические выражения

Они нужны для того чтобы выразить логические суждения. Чаще всего они употребляются в условиях, но ими можно оперировать не только там.

&& – И – AND.

	0	**1**
0	0	0
1	0	1

т.е. оба выражения верны, то выражение верно.

Пример: "Принеси мне молоток **И** гвозди." Если вам принесут и то, и другое, вы будете довольны. В противном случае – недовольны.

|| – Или – OR.

	0	**1**
0	0	1
1	1	1

т.е. если одно из выражений верно, то выражение верно.

Пример: "Принеси мне какую-нибудь стрелялку ***или*** гоночки. Мне всё равно". Вы будете недовольны только, если вам ничего не принесут.

^^ – Исключающее или – XOR.

	0	**1**
0	0	0
1	0	0

т.е. если выражения разные, то общее выражение верно.

Пример: "Принеси мне что–нибудь одно: либо орехи, либо пельмени. А то у меня несварение желудка после пельменей с орехами, а есть охота". Вы будете довольны, когда вам принесут либо то, либо другое. Но если вам ничего не принесут, либо принесут и то, и другое вы будете злы.

Логические выражения могут быть:

Удовлетворение – true – 1 – правда.
Неудовлетворение – false – 0 – ложь.

-А что нужно чтобы сравнить какие-нибудь данные?
А для сравнения нужны операторы:
Операторы бывают 2 типов:
Унарные – это те операторы, которые используют только одно выражение.
Бинарные – это те операторы, которые используют только два выражения.
Операторы сравнения(Бинарные):

>	Больше
>=	Больше либо равно
==	Равно
!=	Не равно
<=	Меньше либо равно
<	Меньше

Пример сравнения: k > m (k больше m), l == 50 (l равно 50).

Для математических расчётов есть свои операторы:

Математические операторы

Бинарные:

+	Плюс, сложение
–	Минус, вычитание
*	Умножение
/	Деление
Div	Целочисленное деление, т.е. оставляет число, остаток выкидывает
Mod	Оставляет остаток от деления. `x mod y = x – (x div y) * y`

Пример: `x + y, x / 2 + 1, 1280 div 3.`
Также можно: `x += 1;` это тоже самое что и `x = x + 1;`
`y *= 7;` ↔ `y = y * 7;`
`z /= 2;` ↔ `z = z / 2;`
`x -= 1;` ↔ `x = x – 1;`

Унарные:
Унарный оператор значит что нужно лишь одно выражение.

!	Не, Переводит **true** в **false**, а **false** в **true**.
-	Отрицает выражение. 59 = –59
~	Отрицает выражение побитно.

Обычное число : $59_{10} = 111011_2$. Число$_{\text{система исчисления}}$.
Побитно отрицаемое: $\sim 59_{10} = 000100_2 = 4_{10}$.

Пример: !50, –50, ~50.
НЕ ПУТАТЬ: (50 – 50) и (–50) в данном случае минусы два разных оператора, делают совершенно разную работу.

Оператор if:

Этот оператор используется для сравнения двух выражений. Здесь используются операторы сравнения. См. выше.
Общий вид:

```
if(<выражение>)
{
    //выполняется это утверждение, если выражение верно
}
```

Или

```
if(<выражение>)
{
    //выполняется это утверждение, если выражение верно
}
else
{
    //выполняется это утверждение, если выражение неверно
}
```

Пример:

```
if(health < 0) //если(жизни < 0)
{
    //Пускаем кровь.
    //счёт +100 очков.
    //проигрываем звук смерти.
}
```

Утверждение можно не обрамлять фигурными скобками, если оно одно.

Пример:

```
if(k != 0)
    k = 0;
```

И немножко про фигурные скобки:

```
if k>0 if kml!=3 if dx=1 k=1
else c-=1
```

Попробуйте разобраться к какому оператору **if** принадлежит **else**. Только чур не жульничать. Попробовать угадать САМОМУ, а потом проверить на компьютере. Угадаете? ☺ Даже программист с большим стажем наколол-ся на этом, вам советую особенно тщательно подойти к форматированию свое кода.

Часто бывает что нужен код который повторяется. Для этого придумали:

Циклы:

Цикл с предусловием: Цикл с предусловием предполагает, что сначала будет верно выражение, затем выполняется цикл пока выражение снова не станет ложным. Т.е. цикл может и никогда не выполнится.

Общий вид:

```
while(<выражение>)
{
    //если выражение верно, то выполняем цикл.
    //выражение должно изменится на неверно, иначе будет вечный цикл.
}
```

Пример:

```
while(!place_free(x, y))
{
    x = random(room_width);
    y = random(room_height);
}
```

Цикл с постусловием: Цикл с постусловием предполагает, что сначала выполняется цикл, хотя бы раз, а затем повторяется пока верно выражение. Т.е. цикл выполнится по крайней мере один раз.

Общий вид:

```
do
{
    //выполняем 1 раз цикл, затем
    //если выражение верно, то выполняем цикл.
    //выражение должно изменится на неверно, иначе будет вечный цикл.
}
until(<выражение>)
```

Пример:

```
do
{
    x = random(room_width);
    y = random(room_height);
}
until(!place_free(x, y))
```

Цикл со счётчиком: этот цикл выполняется n раз. Этот цикл представляет собой дополненный цикл с предусловием. Читайте подробнее о циклах в Wiki.

Общий вид:

```
for(<инициализация>; <условие продолжения>; <операция>)
{
    //выполнение
}
```

Пример:

```
for(i = 0; i < 9; i += 1)
{
    ball[i].x = 32 + 16 * i; //выполнение
}
```

Последнее в списке, но не самое последнее по значению:

Функций:

Общий вид:

```
<имя функций>(<аргумент1>, <аргумент2>, <аргумент3>, ... ,<аргументN>);
```

В **Game Maker** два типа функций:

1) ***Встроенные.*** Их большое количество, всяких и разных. Но если вам этого не достаточно, то ...

2) ***Скриптовые.*** Т.е. написанные вами собственноручно.

Функция, сама по себе, – отдельная программа. Она может иметь несколько аргументов, а может и не иметь вовсе. Функция может возвращать какое–то значение, а может и не возвращать.

-Так зачем же нужны функций?

Функция выполняет роль минипрограммы, которая часто встречается в нашей основной программе. Например, функция по высчитыванию расстояния между точками – будет использоваться вами очень часто. Функций играют очень большую роль в программирований, но в Game Maker не такую как хотелось. Тем не менее, даже в Game Maker они очень полезны, а некоторые незаменимы. Они помогают сократить размер программы, повысить удобность.

Примеры:

```
point_distance(x1, y1, x2, y2); /*возращает расстояние(в пикселях) от точки (x1; y1) до точки (x2; y2).*/
```

Мы не знаем КАК именно эта функция выполняет своё предназначение. И это не важно. Важно то, что она выполняет свою роль и выполняет её «правильно». Теперь мы можем её использовать снова и снова. Где угодно и когда угодно. Написание функций очень не простое занятие, оно требует точного разграничения задачи.

Также функция не может стоять слева от присвоения:

```
instance_nearest(x, y, obj).speed = 0; //нельзя
(instance_nearest(x, y, obj)).speed = 0; /*можно т.к. эта функция возращает ID объекта к которому присваивать можно, но ID должно быть в круглых скобках.*/
```

Основной синтаксис ГМ на этом заканчивается. За более глубокими подробностями отсылаю вас в справку.

Ещё пройдёмся по некоторым наиболее важным и часто применимым математическим функциям:

```
random(x);    //возращает случайное число не более x.
point_distance(x1, y1, x2, y2);    //возращает расстояние между двумя точками.
point_direction(x1, y1, x2, y2);    /*возращает угол направленный из точки (x1; y1) в точку (x2; y2).*/
round(x);    //округляет число x.
min(арг1, арг2, арг3, ... , арг16);    /*возращает мин. значение из всех аргументов. Всего можно 16 аргументов.*/
max(арг1, арг2, арг3, ... , арг16);    /*возращает макс. значение из всех аргументов. Всего можно 16 аргументов.*/
abs(x);    //возращает абсолютное значение x, т.е. без минуса.
sqrt(x);    //возращает корень от x.
lengthdir_x(len, dir);    /*возращает горизонтальную компоненту x от заданного вектора, где len - длина вектора, dir - направление.*/
lengthdir_y(len, dir);    /*возращает вертикальную компоненту y от заданного вектора, где len - длина вектора, dir - направление.*/
```

Геймплей:

Движок ГМ основан на событиях, т.е. каждому действию предшествует событие. Вот основные события:
ev_create – ***Create.*** Вызывается, когда создаётся образец.
ev_destroy – ***Destroy.*** Вызывается, когда образец уничтожается.
ev_alarm – ***Alarm.*** Вызывается, когда будильник "прозвенит".
ev_keyboard – ***Keyboard.*** Вызывается, когда нажимается клавиша на клавиатуре.
ev_mouse – ***Mouse.*** Вызывается, когда нажимается кнопка на мышке.
ev_step – ***Step.*** Вызывается каждый шаг.
ev_collision – ***Collision.*** Вызывается при столкновений с выбранным объектом.
ev_draw – ***Draw.*** Вызывается, после каждого шага, для рисования.
Выполняются они друг за другом, в том порядке в котором они здесь написаны (сверху вниз). Во всяком случае, если верить справке.

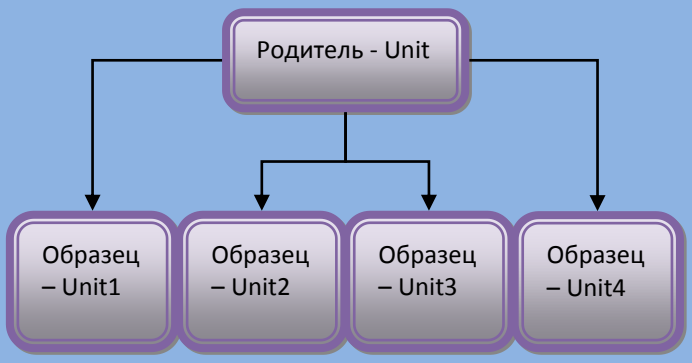
-Все эти события должны применятся к чему–то?
Именно так. Применяются они к объектам. Каждый объект имеет события. Когда происходит событие у этого объекта, исполняется действие, написанное разработчиком, в этом событий. Если мы хотим создать несколько объектов одного типа, то мы их множим. Множим методом **Instance** (образец), т.е. каждый образец независим друг от друга, но если изменить свойство (параметр) объекта–родителя, то изменённое свойство применится ко всем образцам.

```
inst = instance_create(x, y, obj);    /*создаёт образец объекта obj в (x; y) и возвращает его номерок, по этому номерку вы всегда можете контролировать этот образец.*/
inst.speed = 3;    //новому образцу мы задали скорость. Вот так управляются образцы.

instance_destroy();    //уничтожает образец в котором вызвана данная функция.

//а с помощью это конструкций можно удалить другой образец:
with(образец)
    instance_destroy();
//удаляет все образцы
with(all)
    instance_destroy();
```

Объект Unit. Объект–родитель – Нет.
Образец Unit1. Объект–родитель – Unit.
Образец Unit2. Объект–родитель – Unit.
Образец Unit3. Объект–родитель – Unit.
Образец Unit4. Объект–родитель – Unit.



Если изменить скорость Unit1, то изменится скорость только у него. Но...
Если изменить скорость Unit, то она изменится у всех образцов этого объекта, т.е. и у Unit1, Unit2, Unit3 и у Unit4. Видите как направлены стрелки? Т.е. родитель отсылает все данные в образцы, но не наоборот.

Это очень важно. Таким образом, например, если пуля попала в образец, то жизни надо отнимать у этого образца, а не у объекта–родителя.

Адресация:

Чтобы передать какому–то объекту параметр нужно: написать имя объекта, затем поставить точку, написать параметр который хотим изменить, а затем исполнить присвоение нужной величиной.
Допустим мы хотим остановить все образцы объекта Unit, для этого нужно:

```
Unit.speed = 0;
```

Что мы сделали? Мы передали объекту Unit параметр speed равный 0. И все образцы этого объекта получили скорость = 0, т.е. остановили.
А можно передать также параметр не объекту, а образцу:

```
Soldier.speed = 0;
```

Теперь остановиться только этот солдат.

Также есть следующие константы адресаций (почему константы читайте ниже):

- **self** – адресует себе.
- **other** – адресует другому объекту участвующему в событий столкновение.
- **all** – адресует всем образцам.
- **noone** – адресует никому. (звучит странно, но как мы узнаем позже, будет очень полезна)

Чтобы узнать какие значения содержат данные константы:
в событий ***Draw***:

```
draw_text(32, 32, string(self) + '#' + string(other) + '#' + string(all) + '#' + string(noone) + '#' + string(global));
```

Теперь разберёмся какие есть встроенные параметры у объектов:

Объект

`x` – позиция по оси x.
`y` – позиция по оси y.
`xprevious` – предыдущая позиция по оси x.
`yprevious` – предыдущая позиция по оси y.
`xstart` – стартовая позиция по оси x в комнате.
`ystart` – стартовая позиция по оси y в комнате.
`hspeed` – горизонтальная компонента скорости.
`vspeed` – вертикальная компонента скорости.
`direction` – направление (0–360, против–часовой, 0 = направо).
`speed` – скорость (пикселей за шаг).
`friction` – трение (пикселей за шаг).
`gravity` – кол–во гравитаций(пикселей за шаг).
`gravity_direction` – направление гравитаций (270 вниз).

Хочу заметить, что эти параметры будут у каждого объекта, хотите вы того или нет. Если вы захотите создать такой объект, который просто выводит на экран текст, и таких объектов будет много, то всё будет тормозить. А почему? Потому что в этот момент совершаются другие действия, встроенные в **Game Maker**, которые нельзя отключить. В связи с этим, старайтесь создать как можно меньше «пустых» образцов, т.е. тех которые не используют свои параметры. Или объединить все одинаковые функций в один образец.

Образец имеет центр, этот центр обозначается (`x`; `y`), спрайт объекта рисуется в этом центре, совмещая свой центр с центром объекта. Позиция (0; 0) находится в верхнем левом углу комнаты. Перемещая объект вправо, будет увеличиваться координата `x`. Перемещая объект вниз, будет увеличиваться координата `y`. Объект будет двигаться со скоростью `speed` в направлений `direction`. Параметр `friction` уменьшает скорость объекта каждый шаг.

Создание и редактирование объекта

Чтобы создать объект щелкните на синем шарике вверху на основной панели.
У вас появится окно этого объекта.
Name – имя объекта. Помните правило для названий переменных? Для объектов оно тоже действует.
Sprite – спрайт(картинка, изображение) этого объекта. Поговорим чутка о спрайте. Если у вас нет спрайта, тогда вы можете нажать на кнопку Add. И появится окно спрайта. Как рисовать спрайт, я думаю, разберётесь без меня. А вот что нужно знать это группа Origin (начало). Это начало спрайта. Где будет начало спрайта, там будет центр объекта.
Depth – глубина. Какой объект за каким следует в глубину, т.е. по оси Z. Объект с найменьшей глубиной рисуется поверх остальных. Отрицательная глубина разрешена.
Persistent – продолжать существовать. Когда меняется комната, образец данного объекта с вкл. параметром Persistent, останется в игре со своими нетронутыми параметрами. О нужности этого параметра поговорим в следующей книге.
Visible – видимый ли объект.

Далее вы видите 2 колонки:
Левая колонка события, Правая – действия на эти события.
Правый щелчок на левой колонке(колонке событий) выведет меню.
Add Event – Добавить событие.
Delete Event – Удалить событие.
Change Event – Изменить событие. Все действия, которые были, остаются нетронутыми.
Duplicate Event – Скопировать событие. Все действия, которые были, остаются нетронутыми.

Действия:

Как мы уже договорились сверху, действие нас интересует только 1, а именно:

Вкладка ***CONTROL***, группа ***CODE***, левый кубик с документом без стрелочки. Этот кубик будем называть: код(–ом). Типа вставьте код в такое то событие ↔ перетащить этот кубик в правую колонку в указанное событие. При этом у вас обязательно должно быть выделено событие в которое вставляем код. Больше ничего там не надо.

Создайте событие (Add Event) ***CREATE*** и вставьте код.

Вы увидите новое окно. Теперь это окно будет основным для вас. Домом, так сказать. Нас интересуют пока 2 кнопки. Очень важные для вас.

Первая это Зелёная Галочка (). Это означает сохранить всё, что вы там натворили.

Вторая кнопка - где изображены единички и нолики (). Проверяет вашу орфографию. К сожалению логические ошибки не поймает, не надейтесь. Может быть в светлом будущем, когда компьютеры будут самообучающимися.

```
x = sqrt(3-); //скажет что ошибка
x = sqrt(-3);   /*нескажет что ошибка, но вы это узнаете в игре, при глобальном глюке! ☺*/
```

Так что за логикой придётся очень внимательно приглядывать, особенно при большом, развесистом проекте. В данном примере 2 строчки и разглядеть ошибку очень просто.

```
x = sqrt(мотик.скорость - человек.скорость); //всё логично, до первого теста
```

Теперь чуть сложнее, явных чисел нет. Есть 2 скорости. Вроде бы скорость мотика выше чем у человека и всё вроде бы норм. Но вот мотик остановился, а чел нет. В итоге корень извлекается из отрицательного числа, а без комплексного числа мы не можем правильно посчитать это выражение. Теперь ещё чуть сложнее. А вот когда аргумент в функцию приходит откуда–то извне(из другой функций, скрипта), проследить становиться всё труднее.

```
return (max(0, min(sqrt(argument0 + argument1), 100))); /*тоже всё логично, до первого теста*/
```

Как только один из аргументов будет отрицательным и больше чем другой по модулю, то хана. Уже надо проследить,

- А какой аргумент нарушил равновесие?
- А где он находится?

Допустим эта функция не очень распространена и находится «всего» в пяти разных местах. Вот теперь ищите каждое место и проверяйте.

-Как можно избежать этой и других подобных ошибок?

Это достаточно сложно, многие разработчики бьются над этим. Но баги и глюки как были в программах, так и остались. Бывают ошибки фатальные – после которых программа не может дальше продолжать свою работу, бывают ошибки минорные – после которых продолжение возможно, но с какими-то «неправильностями». Теперь перейдём к нашему случаю, как же предотвратить это безобразие? Ответ будет таков: «Всегда думайте головой!».

```
/*вариант 1*/
return (max(0, min(sqrt(abs(argument0) + abs(argument1)), 100)));

/*вариант 2*/
if(argument0 >= 0 && argument1 >= 0)
    return (max(0, min(sqrt(argument0 + argument1), 100)));
else
    return -1;
```

Вариант 1:

Мы насильно превращаем наши аргументы в положительные, таким образом, никаких фатальных ошибок не будет, но будут ошибки логические. Т.е. при не правильных аргументах результат непредсказуем.

Плюсы:

- Фатальных ошибок не будет.

Минусы:

- Возможны логические ошибки, без права на исправление.

Вариант 2:

Мы делаем проверку на «правильность» аргументов (т.е. проверяем что они оба больше или равны нулю), и в случае их не соответствий нашему условию, возвращаем код ошибки (т.к. функция возвращает число 0-100, то

число -1 вполне нормально). А в самой программе выполняем проверку на возвращение этой функцией, если оно -1, то пускай выскочит табличка, что проблема здесь.

Плюсы:

- Фатальных ошибок не будет.
- Можно узнать где проблема.

Минусы:

- Так же возможны логические ошибки.

Да, да! От них не избавится. Функция-то возвращает код ошибки, вместо какого-то значения. Хотя можно сделать так: при коде ошибки вставлять своё правильное значение. Естественно, работать будет, но не так, как надо. Поэтому советую проверять на правильность аргументов, своих переменных – всегда! Может быть это чуть затратнее, но зато меньше геморроя будет при отладке.

Поговорим конкретнее об событиях.

События

Create – это событие нужно, когда создаётся образец. Образец создался, весь код который был в ***Create*** выполнился. Обычно здесь инициализируются переменные, т.е. им присваивают определенное значение.

Например, `health = 100`. т.е. теперь у образца есть переменная `health` со значением `100`. Также здесь создаются сопутствующие объекты и т.д. список можете продолжить сами.

> Итак, теперь самое главное, `health` – это не жизни, это переменная. И у него есть значение. Многие путаются, и при переменной health <= 0 негодуют, что их персонаж не умирает. Это теперь зависит от ВАС. Значение переменной может быть 100, а может 1, а может –1. Но для облегчения будем считать, что health – это жизни. Им можно придать какой–то диапазон. [0; 100]. [0; 1]. [–10; 65535]. И помните, если диапазон [0; 1], это не значит что персонаж умрёт за 1 попадание, и это не значит, что при жизнях меньше 0 персонаж умрёт. Теперь вы ПРОГРАММИСТ. Вы решаете, что и как делать. Всю логику придётся продумывать вам, зависящую от ваших задач и ваших предпочтений.

Step – это событие происходит каждый шаг. Сколько шагов в секунде, задаётся в параметрах комнаты (`room_speed`, по умолчанию =30). Т.е. по умолчанию, код в событий ***Step*** выполняется 30 раз в секунду.

Вот здесь мы можем, например, проверять сколько у нас жизней осталось. По умолчанию, 30 раз в секунду мы проверяем жизни персонажа. Пример:

```
/*При жизнях равных или меньше нуля, убиваем себя.*/
if(health <= 0)
    instance_destroy();   //удаляет этот образец
или
/*При жизнях меньше -10 удаляем себя.*/
if(health < -10)
    instance_destroy();   //удаляет этот образец
или
/*При жизнях меньше нуля, восстановить жизни до предела и создать образец unit.*/
if(health < 0)
{
    health = health_max;
    instance_create(x + 16, y, unit);   //здесь создаём копию
}

/*не забудьте что переменная health встроенная! Поэтому вместо её надо вставить своё название, иначе будете работать с встроенной переменной, а она работает по своим законам.*/
```

А ещё в ***Step***'е можно, например, сделать обработку клавиатуры и мышки:

```
	if(keyboard_check_released(ord('I')))
	{
	    //выполняем действие при отпусканий кнопки I
	    /*вместо ord('I'), введите свой ключ(клавишу) vk_ ... Полный список ключей, пра-
вильно, в справке*/
	}

	if(keyboard_check_direct(vk_space))
	{
	    /*Здесь нажатие кнопки смотрится напрямую с железа. Подробности в справке.
	    При нажатий стреляем*/
	}

	//А можно сделать так:
	k_fire = vk_space;
	if(keyboard_check_direct(k_fire))
	{
	    /*Здесь нажатие кнопки смотрится напрямую с железа. Подробности в справке.
	    При нажатий стреляем*/
	}
	/*теперь мы можем заменить значение k_fire на что угодно, таким образом, перенастроить
клавиши*/

	//мышка работает по такой же технологий. Подробности в справке.
```

COLLISION – в этом событий проверяется столкновение с объектом `room_speed` раз.

Допустим есть 2 объекта: чел и пуля.

Если в объекте чел создано событие ***COLLISION*** с пулей, то адресация идёт **`self`**`.health -= 10;` или просто `health -= 10;`, а если ***COLLISION*** создан в пуле, то **`other`**`.health -= 10;`, потому как мы хотим отнять не у пули жизни, а у другого (**`other`**) объекта, в данном случае чела.

> Кстати, фишка для вас (кстати, не только для новичков):
> Видите я вверху написал везде цифры 10 в отнятий от жизни? Вот. А надо было написать в пуле **`other`**`.health -= damage;` , а в ***CREATE*** пули: `damage = 10;` Таким образом, чтобы подкорректировать урон пули, мне не надо ползать по всему коду и искать эту десятку (если она одна, а если нет?), а только заменить 1 цифру в ***CREATE*** пули. Всё! Поэтому заменяйте все костанты на «константные» переменные.

DRAW – в этом и только в этом событий можно рисовать. Это событие тоже повторяется `room_speed` раз. Написав в это событие хоть что–нибудь, встроенный спрайт объекта теперь не будет отрисовываться. Его надо будет рисовать вручную.

Например, так:

```
	draw_sprite(sprite, -1, x, y);	/*рисуем спрайт sprite, с анимацией, в точке объекта
(x; y)*/
	или так:
	draw_sprite_ext(sprite, -1, x, y, 1, 1, direction, c_white, 1);
	/*рисуем спрайт sprite, с анимацией, в точке объекта (x; y), с масштабом (1; 1), на-
правлением спрайта в направлений объекта
	(direction), со смешиванием с белым цветом(c_white), полностью не прозрачным (1)*/
```

Заметьте: вы можете рисовать какие–угодно спрайты, а не только спрайт этого объекта.

Естественно, есть и другие функций рисования. В справку.

DESTROY – вызывается после уничтожения образца. Здесь можно подчищать за образцом, удалять сопутствующие объекты.

ALARM – вызывается когда время будильника истекает. Всего 12 будильников. Время может быть любое целое, положительное число. Каждый шаг время будильника уменьшается на 1. Когда время будильника равно нулю, тогда событие выполнятся. Время будильника можно поставить так:

`alarm`[номер_будильника] = время_будильника;

По моему, всё! Теперь вы знаете около 10% всех возможностей ГМ. Изучили основы программирования. Теперь надо идти практиковаться. На этом форуме есть множество других статей\примеров\решений.

Прорекламирую свои ☺:

Запись, Загрузка двоичного(бинарного) файла – здесь я рассказываю о работе с файловой системой. Например, если вам понадобится сохранить или загрузить свои собственные сохранения. Если нужен свой формат файла.

Динамические структуры – здесь я рассказываю как можно хранить данные в ГМ. Допустим вы создаёте 2 образца, вы их можете запихнуть в 2 переменные самостоятельно, но как сделать так чтоб можно было ещё создавать кучу образцов и всегда держать их номерки рядом, вы можете это узнать из этой статьи.

Желаю успехов и до свиданья!

Сокращения:
Game Maker – GM – ГМ – Гамак.
ЛКМ – Левая Кнопка Мыши.
ПКМ – Правая Кнопка Мыши.
Дизайн Документ – Диздок.
Instance – инстанс – образец.
Свойство – св–во.
См. – Смотрите.
Кол–во – Количество.