

**ZDoom - скрипты.**  
**Вводное руководство по ACS**  
**для самых маленьких**  
**всего**  
**на тридцати трех страницах!**



**Вы научитесь писать скрипты**  
**всего за пару часов!**  
**Бестселлер!**

(04-12-2011)

Составлено на основе [zdoom.org/wiki](http://zdoom.org/wiki)

© 2011 VladGuardian

# СОДЕРЖАНИЕ

Что такое скрипт?	3
Начали!	3
Комментарии в программе	3
Скрипты	4
- Типы скриптов	
- Запуск скрипта	
- Прерывание скрипта	
- Приостановка скрипта	
- Перезапуск скрипта	5
- Возврат значения из скрипта	
- Активатор скрипта	
Переменные	6
- Различия между глобальными и локальными переменными	
- Глобальные переменные	
- Локальные переменные	
- Типы переменных	
- Подробнее о типе fixed	
- Функции работы со строками	7
- Имена переменных	
Константы	7
- Стандартные константы	
Массивы	8
Функции	8
Печать сообщений на экране	9
- Print	
- PrintBold	
- HudMessage	
- Выбор шрифта для Print/HudMessage	10
- HudMessage - вычисление задержки	
Математические функции	11
- арифметические (+, -, *, /, %, ++, --, abs, sqrt, pow)	
- функции сдвига (<<, >>)	
- функции минимума/максимума (min, max)	
- функции округления (round, floor)	12
- функция случайных чисел (Random)	
- арифметические с фиксированной точкой (FixedMul, FixedDiv)	
- тригонометрические (cos, sin, VectorAngle, distance)	
Время	13
- Задержка	
- Ожидание наступления события	
- Таймер	
Условия. Ветвления. Циклы.	14
- If	
- Switch	
- While	15
- Мертвые циклы	
- For	16
- Прерывание циклов	
- Продолжение циклов	
Информационные функции	17
- Разрешение экрана	
- Информация об игре	
- Информация об уровне	
- Количество монстров/итемов с определенных тэгом	
Актеры. Игроки.	18
- Общая информация об игроке	
- Свойства актора	
- Свойства игрока	19
- Координаты	
- Уровень пола/потолка под/над актором	20
- Скорости	
- Высота глаз актора	
- Углы взгляда	
- Классификация акторов	

- Принадлежность актора определенному классу	21
- Универсальная функция для определения класса актора	
- Перерождение монстров/игроков	
- Смена уровня	
Мультиплеер	22
- Количество подключившихся игроков	
- Номер игрока	
- Проверка участия игрока	
- Фраги игрока	
- Проверка на бота	
Оружие/Инвентарь	23
- Смена оружия	
- Емкость боеприпасов игрока	
- Проверка инвентаря игрока/актеров	
- Очистка инвентаря	
- Отбор предмета инвентаря	24
- Выдача предмета инвентаря	
- Вооружение NPC	
Порождение предметов/монстров/ снарядов	25
- Порождение в точке с координатами	
- Порождение в позиции SpawnSpot-a	
- Порождение снаряда	26
Телепортация	26
- Телепортация предмета	
- Телепортация активатора	
Освещение	27
- Освещенность сектора	
- Цвет освещения сектора	
- Цвет тумана сектора	
- Освещенность актора/игрока	
Экранные пост-спецефффекты	27
- Окрашивание экрана	
- Плавная смена оттенка экрана	
- Смена палитры объектов "на лету"	
Физика	28
- Гравитация	
- Сопротивление воздуха	
- Приложение силы к предмету	
- Землетрясение	
Под водой	28
- Запас воздуха в легких	
Модификация уровня	29
- Двери	
- Задать повреждение для сектора	
- Повредить всем объектам в секторе	
- Повредить активатору скрипта	
- Замена текстур	30
- Небо	
- Зеркальный пол	
- Блокирование проходов	31
- Привязка события к лайндефу	
- Привязка события к сектору (на подъем/опускание сектора)	
Звук и музыка	32
- Музыка	
- Окружающий звук	
- Точечный звук	
Камеры	33
- Через какую камеру смотрит игрок?	
- Отображение вида из камеры на текстуру	
Консоль	34
- Логирование (вывод строк в консоль)	
- Получение значений переменных консоли	
Заключение	34
Приложение	35
- Типы предметов (список констант)	

## Что такое скрипт?

Скрипт – это подпрограмма (или функция, или процедура, как вам удобнее), выполняющая определенную маленькую задачу в вашей карте. Например, выпустить кибердемона из клетки, оздоровить игрока или отобразить на экране количество найденных игроком ключей.

В ZDoom используется ACS (Action Code Script) – скриптовый язык программирования, изначально разработанный для Hexen компанией Raven Software, впоследствии значительно расширенный в ZDoom.

*«Suppose we put a red key in the map. In vanilla Doom, picking it up gives you a red key, enabling you to open red doors or activate swithces that require the red key. Well, with ACS Scripting, you can make any object do any DOOM relating thing you want! You can have a red key kill the player, give him the BFG9000, do absolutely nothing, or raise dead monsters! If you have two red keys, you can make each one do something completely different!»* (zdoom wiki)

Впечатляет, не так ли?

## Начали!

Что необходимо сделать, чтобы начать писать скрипты для своей карты? (допустим, в DoomBuilder уже открыта ваша карта в GZDoom-формате). В DoomBuilder, вы должны нажать F10 (откроется текстовый редактор) и написать следующую строчку:

```
#include "zcommon.acs"
```

Это подключит все необходимое для работы (в том числе и файлы zdefs.acs, zspecial.acs).

Помните, что регистр (размер букв) не имеет значения при написании ключевых слов и имен переменных. Но все же стоит писать ключевые слова *везде* либо с малой буквы, либо везде с большой. У программистов это называется "придерживаться одного стиля".

Также не забывайте комментировать свой код.

## Комментарии в программе

Рано или поздно у вас появится много кода (на несколько страниц), и этот код будет крайне полезно комментировать, чтобы не запутаться в своей же писанине.

Наличие комментариев в программе – признак хорошего программиста. По-крайней мере, аккуратного :)

Причем, как показывает многолетний опыт программирования ☺, начинать писать комментарии надо с самой ранней стадии разработки скрипта, иначе вставлять комментарии в готовый код уже потом – будет просто лень...

Есть 2 типа комментариев:

- комментарий "до конца строки"

```
// This is a comment
```

- комментарий "внутри строки"

```
/* This too */
```

## Скрипты

Синтаксис пользовательского скрипта (так называемый "closed script"):

```
script <номер_скрипта> (void)
{
    ... (ваш код)
}
```

Синтаксис :

```
script <номер_скрипта> <тип_скрипта>
{
    ... (ваш код)
}
```

<Номер\_скрипта> должен находиться в пределах [1..999].

### Типы скриптов

Тип	Кто активирует	Когда (условия срабатывания)
OPEN	World	Сразу после загрузки уровня (лишь однажды)
ENTER	Player	Когда игрок входит на уровень (однажды на игрока и на уровень)
RETURN	Player	Когда игрок вернулся на уровень из хаба (hub)
RESPAWN	Player	Когда игрок респавился в сетевой игре
DEATH	Player	Смерть игрока
LIGHTNING	World	Удар молнии на текущем уровне
UNLOADING	World	После выхода с уровня, до загрузки следующего уровня
DISCONNECT	World	Когда игрок отконнектился от сетевой игры

### Запуск скрипта (ACS\_Execute)

Запустить скрипт на выполнение можно двумя способами:

1) из другого скрипта

```
ACS_Execute (script, map, s_arg1, s_arg2, s_arg3)
```

*Пример:*

```
ACS_Execute(5, 2, 0, 0, 0); // запуск 5-го скрипта на 2-й карте (без аргументов - 0,0,0)
```

2) из карты (событием)

### Прерывание скрипта (terminate)

*Пример: (скрипт прервется, если игрок еще не собрал все ключи)*

```
int keys = 0;

script 1 (void)
{
    keys++;
    if (keys < 3)
        terminate;
    // All keys collected, open the door.
    Door_Open (24, 16, 0);
}
```

### Прерывание других скриптов (ACS\_Terminate)

```
ACS_Terminate(script, map)
```

```
script 51 (void)
{
    Print(s:"The bomb has been defused!");
    ACS_Terminate(17, 0); // прерываем 17-й скрипт на этой карте
}
```

### Приостановка скрипта (suspend)

Интересное применение команды suspend – реализация компьютерного терминала, на котором игрок может пролистывать страницы/экраны по нажатию клавиши Use.

*Пример: (приостановка скрипта до следующего его вызова)*

```
script 1 (void) {
    SetLineTexture (60, SIDE_FRONT, TEXTURE_MIDDLE, "SCREEN2");
    suspend;
    SetLineTexture (60, SIDE_FRONT, TEXTURE_MIDDLE, "SCREEN3");
    suspend;
    SetLineTexture (60, SIDE_FRONT, TEXTURE_MIDDLE, "SCREEN1");
}
```

}

### Приостановка другого скрипта

Приостановить выполнение другого скрипта можно командой:

```
ACS_Suspend(script, map)
```

Детальнее здесь: [http://zdoom.org/wiki/ACS\\_Suspend](http://zdoom.org/wiki/ACS_Suspend)

### Перезапуск скрипта (restart)

*Пример: (выдача игроку бонуса здоровья каждую секунду)*

```
Script 1 ENTER
{
    GiveInventory("HealthBonus",1);
    Delay(35);
    Restart;
}
```

### Возврат значения из скрипта (SetResultValue)

В языках программирования C/C++/C# для возврата значений используется оператор return. Он же используется и в функциях (function) скриптового языка ACS. Но внутри скриптов для возврата значений используется SetResultValue(<значение>).

*Пример: (скрипт #1 напечатает значение 667, возвращенное скриптом #2)*

```
script 1 (void)
{
    Print(d:ACS_ExecuteWithResult(2, 0, 0, 0)); //prints 667
}

script 2 (void)
{
    SetResultValue(667);
}
```

### Активатор скрипта (ActivatorTID)

Активатор скрипта – это объект, запустивший скрипт на выполнение. Активаторами обычно являются монстры или игроки. Узнать, кто запустил скрипт, можно функцией ActivatorTID():

```
script 1 (void)
{
    if (ActivatorTID () == 999)
        Print(s:"You are not a zombie");
    else
        DamageThing(0); // kill it
}

script 10 ENTER
{
    Thing_ChangeTID (0, 999);
}
```

Пояснение: при входе в уровень автоматически запускается скрипт #10 (потому что ENTER). Он присваивает игроку tid=999. Далее где-то в процессе игры Вы вызываете скрипт #1, который проверяет tid вызвавшего объекта. Если tid равен 999, будет выдано сообщение "You are not a zombie", в противном случае объект (монстр) будет убит.

Страшное слово "TID" на самом деле означает "Thing Identifier" (идентификатор предмета).

### Изменение активатора скрипта (SetActivator)

По каким-то причинам может потребоваться изменить активатора скрипта. Эта функция меняет активатора на первого попавшегося субъекта типа tid:

```
int SetActivator(int tid)
```

Функция вернет: 1-в случае успеха, 0-если объектов с тэгом tid не существует.

## Переменные

### Различия между глобальными и локальными переменными

- Глобальные переменные – используются для хранения информации, которая нужна на протяжении всей игры.  
Например: уровень здоровья, кол-во найденных секретов.
- Локальные переменные – используются для хранения информации, нужной лишь внутри какого-то одного скрипта.  
Например:

Первым делом надо *объявить* переменную, иначе получим ошибку:  **Identifier has not been declared.**  
Нет различия в синтаксисе объявления глобальных и локальных переменных. Различие – только в *месте* объявления.

### Глобальные переменные

Глобальные переменные доступны *всем* скриптам и объявляются между строкой `#include` и началом первого скрипта:

```
#include "zcommon.acs"

int mana=0;           // глобальная переменная mana
int found_secrets=0; // глобальная переменная found_secrets

script 1 open // этот скрипт может пользоваться переменными mana и found_secrets
{
    ...
}

script 2 (void) // и этот скрипт тоже
{
    ...
}
```

### Локальные переменные

Локальная переменная доступна только в том скрипте, где она объявлена, и нигде за его пределами:

```
script 2 (void)
{
    // целочисленная переменная (i) объявлена внутри скрипта (локальная)
    int a = 9;
    print(s:"a is " , d: a);
}

script 3 (void) // этот скрипт не увидит переменную (a)
{
    ...
}
```

### Типы переменных

Имеется 4 типа переменных: int, fixed, bool, str и символ(int):

Тип	Что хранит	Для чего лучше подходит
int	Целые числа	Переменные циклов (см. For/While); Счетчики патронов/секретов/монстров;
fixed	Вещественные числа	Координаты любых точек/объектов на карте; Координаты игроков/монстров; Скорость юнита.
bool	Логический тип	Значения TRUE/FALSE (Истина/Ложь)
str	Строки	Уникальные имена объектов ("Generator #5"), монстров ("Imp Fedor"); Информационные сообщения ("You have defeated the Boss!")
int	Символ	Хранение отдельных символов. Используется тип int (что и для целых чисел), поскольку отдельного типа char в ACS нет.

### Подробнее о типе fixed

Традиционно центральные процессоры (CPU) выполняли операции с *вещественными* числами гораздо медленнее, чем с *целыми*. Это давняя история, и хорошие программисты всегда стремились ускорить вычисления с вещественными числами. Один из таких методов – использование чисел с фиксированной точкой (fixed) вместо чисел с плавающей точкой (float, double). Особенно это было актуальным в эпоху ранних процессоров (286, 386), хотя сам тип fixed "изобрели" гораздо раньше, еще до эпохи персональных компьютеров. Не обошли вниманием этот замечательный тип и Джон Кармак с Джоном Ромеро, при разработке своих 3D-движков (Wolfenstein3D, Doom, Quake).

Fixed – тип числа с фиксированной точкой, созданный *специально* для ускорения вычислений в CPU. Он занимает те же 4 байта, что и тип float, однако вычисления с ним идут в 3-4 раза быстрее. Однако чем-то мы должны пожертвовать ради скорости.

Мы неизбежно жертвуем:

- 1) *точностью* – невозможно записать число точнее чем  $\pm 0.000015258789$  ( $= 1/65536$ ), но этого достаточно для игр.

2) диапазоном представимых чисел. Числа вне диапазона (-32767..32768) невозможно записать в этом формате. Вот так выглядит байтовая структура fixed-числа:

<Старшие 2 байта> <Младшие 2 байта>

Целая часть	Дробная часть
-------------	---------------

Чтобы преобразовать fixed → вещественное число, надо поделить fixed на 65536.

Чтобы преобразовать вещественное число → fixed, надо умножить вещ.число на 65536.

Примеры:

Число 1.5 в виде fixed записывается как  $1.5 * 65536 = 98304$ .

Fixed-число 205887 представляет число Pi ( $205887/65536 = 3,141586$ ).

## Функции работы со строками

- Длина строки

```
int StrLen(str string)
```

Пример:

```
str st = "This is the string.";
Print (s:st, i:strlen(st)); // печатаем строку и ее длину в символах
```

- Получение символа строки

```
GetChar(str string, int index)
```

Первый символ строки имеет индекс 0, второй - 1. Такова специфика не только ACS, но и мощных "индустриальных" языков C/C++/C#, на которых основаны "серьезные" программы, в том числе, сам ZDoom. Так что в определенном смысле можно гордиться такими "непонятками", как первый элемент, который нумеруется... нулем. И в целом ACS очень сильно напоминает C/C++, что не может не радовать "seasoned programmers".

## Имена переменных

Человеку, стремящемуся как можно быстрее завершить свою карту под ZDoom/GZDoom, и не обращающему внимания на мелкие нюансы, это покажется занудством, но все же прислушайтесь к моим советам:

- легкость восприятия скрипта программистом напрямую зависит от **удачно подобранных имен переменных**
- старайтесь давать **короткие, но понятные** имена переменным: *num\_monsters*, *keys\_found*
- **разделяйте слова** в имени: *num\_monsters* читабельнее, чем *nummonsters*
- не делайте **слишком коротких** сокращений: *n\_k* (кто через месяц вспомнит, что это "number of keys"?)
- не давайте **бессмысленных** имен: *myArray* (мой массив). Массив чего?, *ccc* (что "ccc"?)
- хорошо владеете английским? Давайте **названия из английских слов**, а не из транслита-суржика: *num\_monsters* очевидно предпочтительней, чем *kolvo\_monstrov* (еще и избежите насмешек коллег "по цеху")
- **переменным циклов** дают обычно очень **короткие имена** из этих букв (i, j, k)
- не забывайте – **регистр букв** не имеет значения. Имя переменной *boss* – эквивалентно *Boss*, эквивалентно *BOSS*. Если объявить в одном месте обе (или три) таких переменные, компилятор просто ругнется о повторном объявлении переменной:  **boss: Redefined identifier**

## Константы

В дополнение к переменным, для значений, которые НЕ будут изменяться внутри скриптов, лучше применять константы:

```
#define <имя_константы> <значение>
```

Константы могут быть числовые, строковые и логического типа. Обратите внимание – ключевых слов int/str/bool нет, компилятор ACS сам определит, что ему "подсунули" в #define:

Примеры:

```
#define MAX_OBJECTIVES 4 //кол-во заданий на карте
#define LEVEL_NAME "*Imps Canyon*\nby Vasya Malinin" //my level name
#define RELEASE_VERSION TRUE //это release-версия моей карты
```



Где только возможно, используйте константы вместо "магических чисел".

Например, запись:

```
SetPlayerProperty(1, 1, 4); // заморозить игрока
```

непонятна: что, например, означает "4"? Вот та же команда, параметры которой записаны константами:

```
SetPlayerProperty(1, ON, PROP_TOTALLYFROZEN); // заморозить игрока
```

## Стандартные константы

Подключив к своему скрипту:

```
#include zcommon.acs
```

вы автоматически получаете набор полезных констант (из файла zdefs.acs)

: ON / OFF, YES / NO, TRUE / FALSE, SKILL\_VERY\_EASY / SKILL\_EASY / SKILL\_NORMAL / SKILL\_HARD / SKILL\_VERY\_HARD и множество других.

## Массивы

Массив – полезная штука, когда нужно хранить данные о нескольких объектах сразу (игроках, например):

```
int myArray[200]; // массив без инициализации элементов
int anotherArray[3] = {3,6,9}; // массив с инициализацией элементов
```

Нумерация элементов начинается с нуля: первый элемент имеет индекс "0" (monster[0]), второй – индекс "1" (monster[1]) и т.д. Это кажется неудобным только на первый взгляд...

Массив может быть любого типа: int, float, str или bool.

Однако из-за слабой проверки типов в ACS, становится возможным объявить массив смешанных типов:

```
int monster[2] = {"DoomImp", 5, 98304};
```

И хотя массив явно объявлен как тип int, такой массив будет хранить три различных типа данных! В этом легко убедиться, распечатав содержимое:

```
print(s:monster[0], s:" ", i:monster[1], s:" ", f:monster[2]);
```



Третий элемент массива мы трактовали как тип fixed (f:), а fixed(98304) соответствует числу 1.5 (98304 / 65536 = 1.5).

## Функции

Хорошей (правильной) альтернативой скриптам, которые должны возвращать *значение*, являются функции:

*Шаблон функции:*

```
function type function_name([type arg1 [, type arg2 [...]])
{
    // вычисления value
    return value;
}
```

То есть пишете ключевое слово *function*, за ним указываете тип возвращаемого функцией значения *type* (это может быть любой из типов: int, fixed, str, bool или void), далее даете любое имя своей функции (вместо *function\_name*), и далее в скобках перечисляются передаваемые параметры (если параметров нет, в скобках пишем слово *void*).

Открываете фигурную скобку { и сразу же закрываете } (чтобы не забыть этого сделать потом ☺). То, что будет находиться внутри этих скобок, называют *телом функции* (function body).

Если все вышесказанное показалось страшным и непонятным, взгляните на этот простой пример:

*Пример – вызов функции возведения в квадрат:*

```
script 1 open
{
    int ret;
    ret = square(3);
    print(s:"the square of 3 is ", d:ret);
}

function int square(int val)
{
    return val * val;
}
```

Если функция не должна ничего возвращать, используйте слово *void*, в этом случае *return* тоже не нужен:

```
// печатаем кол-во убитых монстров (к общему кол-ву монстров на уровне)
function void monsters_count (int kills, int num_monsters)
{
    print(s:"Kills: ", i:kills, c:'/', i:num_monsters);
}
```

*return* используется только в функциях, в скриптах используйте *terminate* (с предварительным вызовом *SetResultValue*, если из скрипта нужно вернуть значение).

Если попытаться вставить *return* в скрипт, вы получите ошибку:  Return can only be used inside a function.



Вы не можете использовать функцию задержки (delay) в функциях. Если требуется вставить задержку, это надо сделать в *скрипте*, вызвавшем функцию, *до* или *после* вызова функции (как удобно).

Если же вы будете упрямяться, то получите сообщение:  Latent functions cannot be used inside functions.

## Печать сообщений на экране

Для вывода сообщений используются две функции: Print (простая) и HudMessage (мощная).

### Print

```
print (s:"blah-blah");
```

Внутри вызова print можно сочетать разные типы данных, например, строки с числами. Чтобы сделать это, надо указать формат подаваемых данных:

i:	Целое число (i:45)
f:	Число с фиксированной точкой (f:0.5)
s:	Строка (s:"tatata")
c:	Один символ (c:'')
n:	В зависимости от идущего далее числа: n:-1 – имя уровня ("Entryway" в Doom 2) n:-2 – имя карты ("MAP01" в Doom 2) n:-3 – название уровня сложности ("Ultra-Violence") n:0 – имя активатора (имя класса, вызвавшего скрипт, или имя игрока, если вызывал он) n:<положительное_число> - имя N-го игрока в сетевой игре (n:2 – имя 2-го игрока)
k:	Печатает имя клавиши, которая назначена для определенного действия (k:"+use") Удобно, как надо дать игроку подсказку, какую клавишу нажать.

Полный список форматов можно посмотреть тут: [http://zdoom.org/wiki/Print#Cast\\_type](http://zdoom.org/wiki/Print#Cast_type)

Если не указать тип подаваемых данных, вылезет ошибка:  Unknown cast type in print statement.

Пример:

```
Print(s:"I need ", d:x, s:" shells"); // переменная x хранит число патронов к дробовику
```

Внутри строки, кроме обычных алфавитно-цифровых символов, можно использовать спецсимволы:

\n	Переход на новую строку
\xXX	Шестнадцатеричный символ с кодом XX
\cC	Подкрашивание текста цветом с кодом C

Цвета для print (коды):

code	color	code	color	code	color	code	color
a	 reddish brick	g	 red	n	 light blue	t	 purple
b	 tan	h	 blue	o	 cream	u	 dark gray
c	 gray	i	 orange	p	 olive	v	 cyan
d	 green	j	 white	q	 dark green	l	Use original colors
e	 brown	k	 yellow	r	 dark red	-	Default color for Print messages
f	 gold	m	 black	s	 dark brown		

Пример печати расцвеченной строки:

```
print(s:"\cgRoses are red\n\chViolets are blue\n\cjSilver for me\n\cfGold for you");
```



### PrintBold

PrintBold – совершенно аналогичен Print, только сообщение увидят все игроки в сетевой игре.

### HudMessage (улучшенный print)

HudMessage – мощный аналог команды print.

У нее есть 4 разновидности:

- 1) простой текст
- 2) текст с затуханием
- 3) текст с появлением и затуханием
- 4) печатающийся текст
- 5) вывод картинки

1) Простой текст:

```
hudmessage (text; HUDMSG_PLAIN, int id, int color, fixed x, fixed y, fixed holdTime);
```

2) Текст с затуханием:

```
hudmessage (text; HUDMSG_FADEOUT, int id, int color, fixed x, fixed y, fixed holdTime, fixed fadetime);
```

3) Текст с появлением и затуханием:

```
hudmessage (text; HUDMSG_FADEINOUT, int id, int color, fixed x, fixed y, fixed holdTime, fixed inTime, fixed outTime);
```

4) Печатающийся текст:

```
hudmessage (text; HUDMSG_TYPEON, int id, int color, fixed x, fixed y, fixed holdTime, fixed
typetime, fixed fadetime);
```

### 5) Вывод картинки:

```
SetFont("PICTURE");
// в строке должен быть только один символ - "A"
HudMessage(s:"A"; HUDMSG_PLAIN, int id, int color, fixed x, fixed y, fixed holdTime);
```

id – идентификатор сообщения. Используйте, чтобы два подряд идущих сообщения не перекрывали друг друга.  
color – цвет; x,y – координаты на экране;  
holdTime – время задержки на экране; typeTime – время печатания одной буквы; fadeTime – время затухания.

#### Диапазон координаты X:

[0.0, 1.0]: Position between left and right edge valid box locations  
[-1.0, 0.0]: Position between left and right edge of screen  
(1.0, 2.0): Same as [0.0,1.0], but center each line inside box  
[-2.0, 1.0]: Same as [-1.0,0.0], but center each line inside box

#### Диапазон координаты Y:

[ 0.0, 1.0]: Position between top and bottom of valid box locations  
[-1.0, 0.0]: Position between top and bottom edge of screen

#### Примеры координат (x, y):

(0.5, 0.0)	Сообщение сверху экрана, посередине (по горизонтали).
(0.0, 0.5)	Сообщение слева экрана, посередине (по вертикали).
(-0.25,0.0)	Сообщение сверху, на 1/4 выходит за левую кромку экрана.
(1.5, 0.5)	Центрирует прямоугольник вывода на экране, также центрирует каждую строку внутри прямоугольника

#### Цвета для HudMessage (коды):

Обычные цвета			Особые "цвета"
CR_BRICK 0	CR_RED 6	CR_CREAM 14	CR_UNTRANSLATED -1
CR_TAN 1	CR_BLUE 7	CR_OLIVE 15	
CR_GRAY 2	CR_ORANGE 8	CR_DARKGREEN 16	
CR_GREY 2	CR_WHITE 9	CR_DARKRED 17	
CR_GREEN 3	CR_YELLOW 10	CR_DARKBROWN 18	
CR_BROWN 4	CR_BLACK 12	CR_PURPLE 19	
CR_GOLD 5	CR_LIGHTBLUE 13	CR_DARKGRAY 20	

#### Выбор шрифта для Print/HudMessage

```
SetFont("SMALLFONT"); // мелкий (обычный) шрифт
SetFont("BIGFONT"); // крупный шрифт
SetFont("CONFONT"); // такой же шрифт, как и в консоли
```



*На заметку: CONFONT хоть и мелкий, но в этом шрифте разные по размеру большие и малые буквы (в отличие от SMALLFONT и BIGFONT).*

#### HudMessage - вычисление задержки (HudMessageTime)

После вывода текста, как правило, нужно на некоторое время оставить текст на экране, чтобы игрок успел его прочесть. Подстановка длительности задержки наобум обычно не дает хороших результатов. Авторами ZDoom разработана специальная функция HudMessageTime, которая вычисляет величину задержки автоматически, исходя из количества символов строки, времени появления и угасания:

```
#define TICUNIT 35.725 // количество "тиков" в одной секунде
function int HudMessageTime(int type, int length, int typetime, int staytime, int fadetime)
{
    Switch(type) {
    Case HUDMSG_PLAIN: return fixedmul(staytime, TICUNIT) >> 16;
    Case HUDMSG_FADEOUT: return fixedmul(staytime + fadetime, TICUNIT) >> 16;
    Case HUDMSG_TYPEON:
        return fixedmul(fixedmul(typetime, length << 16) + staytime + fadetime, TICUNIT) >> 16;
    Case HUDMSG_FADEINOUT: return fixedmul(typetime + staytime + fadetime, TICUNIT) >> 16;
    }
    return 0;
}
```

Функцию HudMessageTime нужно вызывать *внутри* функции delay.

*Пример: (при старте уровня показываем 10 строк из массива Strings с задержкой после показа каждой строки)*

```
script 1 ENTER
{
    delay(35);
    for (int i=0; i<10; i++) {
        HudMessage(s:Strings[i]; HUDMSG_TYPEON, 0, CR_RED, 1.5, 0.8, 2.0, 0.1, 0.5);
        delay(HudMessageTime(HUDMSG_TYPEON, strlen(Strings[i]), 0.1, 2.0, 0.5);
    }
}
```



## Математические функции

- арифметические функции (+, -, \*, /, %, ++, --, abs, sqrt, pow)
- функции сдвига (<<, >>)
- функции минимума/максимума (min, max)
- функции округления (round, floor)
- функция случайных чисел (Random)
- арифметические функции с фиксированной точкой (FixedMul, FixedDiv)
- тригонометрические функции (cos, sin, VectorAngle, distance)

### Арифметические функции

+ - *	Здесь комментариев не требуется.
/	Осторожно с функцией деления! Деление целых чисел происходит с отбрасыванием дробной части и округлением до меньшего целого! Таким образом, $5/4 = 1$ (а не 1.25 :)
%	Остаток от деления. Примеры: $9\%17 = 8$ , $5\%2 = 1$ , $4\%2 = 0$ .
++, --	Прибавление/вычитание единицы к/от переменной. Эквивалентно $a=a+1$ , $a=a-1$ , но лучше пользоваться этим вариантом. Работает быстрее.
abs	К сожалению, в скриптах ACS не предусмотрено функции модуля числа (abs). Но вы можете обойти это недоразумение, вставив в скрипт свою собственную функцию: <pre>function int abs (int x) {     if (x &lt; 0)         return -x;     return x; }</pre>
sqrt	К сожалению, в скриптах ACS не предусмотрено и функции квадратного корня (sqrt). Но имеется сразу несколько эмуляций этой функции, работающих с разной точностью/скоростью: <a href="http://zdoom.org/wiki/Sqrt">http://zdoom.org/wiki/Sqrt</a> - выбирайте любую.
pow	Такой функции, как возведение в целую степень, тоже нет. Вот замена: <pre>function int pow (int x, int n) {     int y = 1;     while (n-- &gt; 0) y *= x; // умножаем (x) n-ное число раз само на себя     return y; }</pre>

### Функции сдвига

<<	Битовый сдвиг влево аналогичен умножению на 2 в степени (n), n – величина сдвига. Сдвиг на 1 бит влево: $(X \ll 1)$ аналогично $(X*2)$ // * 2 <sup>1</sup> Сдвиг на 2 бита влево: $(X \ll 2)$ аналогично $(X*4)$ // * 2 <sup>2</sup> Сдвиг на 3 бита влево: $(X \ll 3)$ аналогично $(X*8)$ // * 2 <sup>3</sup> Сдвиг на 4 бита влево: $(X \ll 4)$ аналогично $(X*16)$ // * 2 <sup>4</sup> Сдвиг на 5 битов влево: $(X \ll 5)$ аналогично $(X*32)$ // * 2 <sup>5</sup> ...
>>	Битовый сдвиг вправо аналогичен делению на 2 в степени (n), n – величина сдвига. $(X \gg 1)$ аналогично $(X / 2)$ // 2 <sup>1</sup> $(X \gg 2)$ аналогично $(X / 4)$ // 2 <sup>2</sup> $(X \gg 3)$ аналогично $(X / 8)$ // 2 <sup>3</sup> $(X \gg 4)$ аналогично $(X / 16)$ // 2 <sup>4</sup> $(X \gg 5)$ аналогично $(X / 32)$ // 2 <sup>5</sup> ...

Довольно странные операции – эти сдвиги, не правда ли? Почему бы не пользоваться обычным умножением и делением? Трудно дать совет, когда и как пользоваться сдвигом, скорее всего, вам в нужный момент это подскажет интуиция. Просто помните о том, что есть такие – *операции сдвига*.

### Функции минимума/максимума

min	<pre>function int min (int a, int b) {     if (a &lt; b)         return a;     return b; }</pre> <p>Примечание: если a и b равны, функция вернет b.</p>
max	<pre>function int max (int a, int b) {     if (a &gt; b)         return a;     return b; }</pre> <p>Примечание: если a и b равны, функция вернет b.</p>

## Функции округления

round округляет до ближайшего целого, floor просто отбрасывает дробную часть.

round	<pre>function int round(int fixedNumber) // Returns integer value {     return (fixedNumber + 0.5) &gt;&gt; 16; }</pre>
floor	<pre>function int floor(int fixedNumber) // Returns fixed point value {     return fixedNumber &amp; 0xFFFF0000; }</pre>

## Функция случайных чисел

random	<p>Возвращает случайное число в диапазоне [min..max] (включительно).</p> <pre>int Random(int min, int max)</pre> <p><i>Пример: (повреждение объекта случайным уроном в пределах [1..10])</i></p> <pre>DamageThing( Random(1, 10) );</pre> <p>Можно сколько угодно спорить о "(не-)трусости" функции random, о равномерности ее распределения, но трудно НЕ согласиться с тем, что для большинства "думовских" применений ее будет достаточно.</p>
--------	---

## Арифметические функции с фиксированной точкой

Нельзя напрямую (\*) перемножать или делить числа с фиксированной точкой, это дает неверный результат:

```
Print (f: 0.5 * 0.5); // 16384, oops!
Print (f: 1.0 / 0.5); // 0.000030515, WTF?
```

Для этого созданы специальные функции:

```
FixedMul(int a, int b)
FixedDiv(int a, int b)
```

*Пример:*

```
Print (f: FixedMul (0.5, 0.5)); // 0.25 - вуаля!
Print (f: FixedDiv (1.0, 0.5)); // 2 - ok
```

Однако, умножение/деление числа с фиксированной точкой на *целое* число дает правильный результат:

```
fixed z = 1.2*3; // пробуем умножить
print(f:z); // 3.6 - ok!
z = 1.0 / 3; // пробуем делить
print(f:z); // 0.333 - ok!
```

## Тригонометрические функции

Тригонометрия жизненно важна для реализации таких вещей, как: вычисление расстояний, углов, взаимного расположения объектов, рисование вручную карты уровня и пр.

cos, sin	<pre>int cos(int angle) int sin(int angle)</pre> <p>Эти функции принимают fixed-угол (90° здесь равны 0.25, а 360° = 1.0), и возвращают значение в формате fixed.</p>
VectorAngle	<pre>int VectorAngle(int x, int y)</pre> <p>Функция принимает вектор (x,y), а возвращает угол формате fixed. Углы отсчитываются от восточного направления и против часовой стрелки (при виде сверху). То есть: восток-0° (fixed-0.0), север-90° (fixed-0.25), запад-180° (fixed-0.5), юг-270° (fixed-0.75).</p>
distance	<pre>function int distance (int tid1, int tid2) {     int x, y, z, d;     x = GetActorX(tid1) - GetActorX(tid2) &gt;&gt; 16; // Convert fixed point to int     y = GetActorY(tid1) - GetActorY(tid2) &gt;&gt; 16;     z = GetActorZ(tid1) - GetActorZ(tid2) &gt;&gt; 16;     d = sqrt( x*x + y*y + z*z );     return d; }</pre> <p>Быстрый вариант функции distance, возвращающий к тому же более точный fixed (вместо int):</p> <pre>function int fdistance (int tid1, int tid2) {     int len;     int y = getactory(tid1) - getactory(tid2);     int x = getactorx(tid1) - getactorx(tid2);     int z = getactorz(tid1) - getactorz(tid2);</pre>

```

int ang = vectorangle(x,y);
if ((ang+0.125)%0.5) > 0.25) len = fixeddiv(y, sin(ang));
else len = fixeddiv(x, cos(ang));

ang = vectorangle(len, z);
if ((ang+0.125)%0.5) > 0.25) len = fixeddiv(z, sin(ang));
else len = fixeddiv(len, cos(ang));

return len;
}

```

## Время

### Задержка

```
delay(tics)
```

Очень простая команда, но вот беда – пауза указывается в каких-то тиках. Чтобы указать секунды, надо просто умножить секунды на 35 (в одной секунде 35 тиков, соответственно один тик – 1/35 секунды):

```
delay(35*sec);
```

"Магическая" команда `delay(1)` (задержка на 1 тик) активно применяется для предотвращения "залипания" цикла с неизбежным прерыванием его системой:

```

script 1 OPEN
{
    while (TRUE) // вечный цикл
    {
        Print(s:"You are playing: ", i:Timer()/35, s:" seconds.");
        delay(1); // залипания не будет
    }
}

```

А такой скрипт после запуска уровня неизбежно вызовет ошибку: **"Runaway script 1 terminated"**

```

script 1 OPEN
{
    while (TRUE) // вечный цикл
    {
        Print(s:"You are playing: ", i:Timer()/35, s:" seconds.");
    }
}

```

### Ожидание наступления события

- Задержка скрипта до момента окончания движения сектора

```
TagWait(int tag)
```

- Задержка скрипта до момента завершения другого скрипта

```
ScriptWait(int script)
```

*Пример: (открытие двери, ожидание завершения ее поднятия, и вывод соответствующего сообщения)*

```

script 1 (int sector)
{
    PrintBold (s:"Opening the hangar doors...");
    Door_Open (sector, 5, 0);
    TagWait (sector);
    PrintBold (s:"Hangar doors now open!");
}

```

### Таймер

Сколько времени прошло с момента запуска уровня?

```

int t = Timer() / 35; // разделив на 35, получили кол-во секунд
HudMessage(d:t/60, s:":", d:(t%60)/10, d:t%10; HUDMSG_PLAIN, 1, CR_RED, 0.95, 0.95, 2.0);

```

## Условия. Ветвления. Циклы.

« 90% of beginners don't need to know how to declare variables or how to use program control statements like 'if' statements. »  
(ZDoom wiki)

Операторы условий/циклов вам пригодятся, рано или поздно. Особенно если вы решили "забабахать" интересный сюжетный вад, с мощной скриптовой основой.

Вам повезло, в отличие от большинства других языков, операторов условий/ветвлений здесь всего три:

- 1) Условие (ветвление): If/Else
- 2) Селектор (ветвление): Switch
- 3) Условный цикл: While
- 4) Цикл со счетчиком: For

### If

*Пример:*

```
if (w<1024)
    print(s:"Recommended resolution is 1024x768.");
```

Идет проверка переменной (w), если она оказалась меньше 1024, вывелось предупреждение.

*Более сложный пример:*

```
if(cond1)          // если условие cond1 верно ...
{
    command1;      // ... будут выполнены эти команды
}
else if(cond2)     // иначе - если условие cond2 верно ...
{
    command2;      // ... будут выполнены вот эти команды
}
else               // иначе (условия cond1 и cond2 были неверны) выполняются команды command3
{
    command3;
}
```

### Switch

Великолепная мощная команда ветвления, как бы унаследованная от языка С – switch.

У нее довольно громоздкий синтаксис, но овчинка стоит выделки.

*Пример: (одним оператором разруливаем разную конфигурацию монстров на уровне на разных уровнях сложности)*

```
switch (GameSkill ()) // анализируем выбранный уровень сложности
{
    case SKILL_VERY_EASY:
        SpawnSpot ("ZombieMan", 60); // SpawnSpot - команда для порождения монстров
        break;

    case SKILL_EASY:
        SpawnSpot ("DoomImp", 60);
        break;

    case SKILL_HARD:
        SpawnSpot ("BaronOfHell", 60);
        // break специально не использован, чтобы выполнялся блок SKILL_NORMAL (ниже)

    case SKILL_NORMAL:
        SpawnSpot ("DoomImp", 61);
        break;

    case SKILL_VERY_HARD:
        SpawnSpot ("Cyberdemon", 60);
        break;
}
```

Обратите внимание – в конце одного из case не используется break, т.к. в случае SKILL\_HARD нам нужно, чтобы кроме BaronOfHell, породился еще и DoomImp.

Если внутри case вы выходите из скрипта (return) выполняет выход из функции *до того*, как будет достигнут следующий Case.



*Оператор switch в принципе может быть записан только с помощью операторов if / else. Хотя лучше применять switch, там где это возможно. См. пример ниже.*

*Пример: (переписываем switch, используя только if/else)*

```
int game_skill = GameSkill(); // присвоим переменной, чтобы много раз не вызывать GameSkill()

if (game_skill==SKILL_VERY_EASY)
    SpawnSpot ("ZombieMan", 60);

else if (game_skill==SKILL_EASY)
    SpawnSpot ("DoomImp", 60);

else if (game_skill==SKILL_HARD)
{
    SpawnSpot ("BaronOfHell", 60);
    SpawnSpot ("DoomImp", 61);
}
else if (game_skill==SKILL_NORMAL)
    SpawnSpot ("DoomImp", 61);

else if (game_skill==SKILL_VERY_HARD)
    SpawnSpot ("Cyberdemon", 60);
```

## While

Синтаксис:

```
While (<условие>)
{
    <тело цикла>
}
```

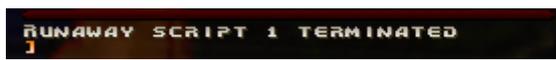
*Пример – while:*

```
int i = 5;
while (i > 0) //выход из цикла произойдет только когда i достигнет нуля (не выполнится (i>0))
{
    <команда (или команды), которые будут уменьшать переменную i>
}
```

## Мертвые циклы

В отличие от оператора if/else, while – довольно опасная команда. В каком смысле? Если долго НЕ наступает условия завершения while, цикл "зацикливается" (каламбур, но и суровая реальность). Такой цикл называют "мертвым циклом". Причем причиной такого поведения цикла может стать не только ошибка программиста в написании скрипта, но и исходные данные уровня, свойства объектов, монстров и т.д.

Например: ваш while ждет, пока освещенность некоего сектора опустится до определенного уровня (станет равным ему), но сектор уже (на момент входа в цикл) имеет освещенность *ниже* этого значения. Такой цикл никогда не закончится. Он будет бесконечно выполняться, не давая работать всей остальной системе ZDoom (и другим скриптам, в частности). В итоге, когда кол-во выполненных команд внутри такого цикла превысит 500 тысяч, цикл будет принудительно прерван системой:

 (сообщение в консоли)

Поэтому потенциально опасные циклы while (а также for, о котором пойдет речь ниже) всегда нужно разбавлять вызовом delay(1), иначе цикл. Что в итоге приведет к прерыванию скрипта.

Анекдот в тему: «*Давай сначала съедим твое, а потом – каждый свое*».

Не думайте, что вставка "магической" строчки delay(1) "вылечит" неправильно написанный цикл. Это предотвратит лишь прерывание цикла, но сам цикл будет гонять по кругу бесчисленное число раз.

Игру это вряд ли сильно затормозит (delay(1) спасает), но ваш скрипт-то будет выполняться неправильно!

*Пример: (мертвый цикл, не вызывающий ошибки runaway)*

```
while (TRUE)
{
    ... (что угодно можете здесь написать, из цикла вам все равно не выйти...)
    delay(1); // "таблетка" от прерывания цикла системой
}
```

*Пример: (мертвый цикл, вызывающий ошибку runaway)*

```
while (TRUE)
{
    ... (что угодно можете здесь написать, из цикла вам все равно не выйти...)
}
```

Подчеркнем, что TRUE эквивалентно таким выражениям:

```
While (1)
While (4==4)
```

Все эти выражения истинны (имеют значение TRUE).

В свою очередь, выражения FALSE, 0, (1==2) – ложны.

## For

*Пример: (содержимое цикла в фигурных скобках выполнится ровно 10 раз)*

```
for (int i=0; i<10; i++) // i = 0,1,2,3,4,5,6,7,8,9 (10 раз)
{
    ...
}
```

Синтаксис цикла состоит из трех частей:

- инициализатора цикла (i=0)
- условия выполнения цикла (i<10)
- счетчика цикла (i++)

Эти части должны быть разделены символом ; иначе не избежать недвусмысленной ошибки:  **Missing semicolon.**

*Пример цикла с убывающим счетчиком:*

```
for (int i=9; i>=0; i--) // i = 9,8,7,6,5,4,3,2,1,0 (10 раз)
{
    ...
}
```

## Прерывание циклов While/For/Switch (break)

Вы замечали раньше странный оператор *break*? Он служит для прерывания циклов While/Switch, а также For. Когда по логике скрипта именно в *этом* месте нужно прервать выполнение цикла, используйте *break*. Куда же переходит управление при вызове *break*? Очень просто – на первый оператор после закрывающей скобки.

*Пример: (где управление после вызова break переходит на тот Print, сразу за циклом)*

```
int count=0;
while (TRUE)
{
    if (count==1000) // ну всё, с меня хватит,
        break;      // прерываем цикл!
    count++; // увеличить счетчик
    delay(1);
}
Print("Slava Bogu, vyrvalis iz okruzhenia! Ura!");
```



***break* может использоваться только в циклах While/For/Switch.**

## Продолжение циклов While/For (Continue)

Что делает *continue*? Игнорирует остаток цикла и переходит к следующей итерации цикла. Если это сложно для понимания, взгляните на пример.

*Пример: (обработка только акторов являющихся монстрами, и игнорирование остальных типов акторов)*

```
for (int i=0; i<10; i++) // анализировать будем 10 штук акторов
{
    if (ClassifyActor(1000+i) != ACTOR_MONSTER) // если актер не является монстром,
        continue; // ... переход к следующему i
    ... // обработка монстра
}
```



**В отличие от *break*, *continue* может использоваться только в циклах While/For.**

## Информационные функции

### Разрешение экрана (GetScreenWidth / GetScreenHeight)

```
script 1 (void)
{
    int w = GetScreenWidth();
    if (w<1024)
        print(s:"Recommended resolution is 1024x768.");
}
```

### Информация об игре

- Сложность игры

```
int GameSkill()
```

Возвращаемые значения: SKILL\_VERY\_EASY, SKILL\_EASY, SKILL\_NORMAL, SKILL\_HARD, SKILL\_VERY\_HARD.

- Тип игры

```
int GameType()
```

Возвращаемые значения: GAME\_SINGLE\_PLAYER(0), GAME\_NET\_COOPERATIVE(1), GAME\_NET\_DEATHMATCH(2), GAME\_TITLE\_MAP(3).

*Пример – предупреждаем игрока, если карта запущена не в режиме deathmatch:*

```
script 1 ENTER
{
    if (GameType () != GAME_NET_DEATHMATCH)
        Print (s:"This is a deathmatch only map!");
    else
        Print (s:"BobDM1\nBy Bob");
}
```

### Информация об уровне

```
int GetLevelInfo(int level_info)
```

level\_info:

LEVELINFO_PAR_TIME	"Par time" уровня
LEVELINFO_CLUSTERNUM	Номер текущего кластера
LEVELINFO_LEVELNUM	Номер карты
LEVELINFO_TOTAL_SECRETS	Общее кол-во секретов
LEVELINFO_FOUND_SECRETS	Кол-во найденных секретов
LEVELINFO_TOTAL_ITEMS	Общее кол-во предметов
LEVELINFO_FOUND_ITEMS	Кол-во найденных предметов
LEVELINFO_TOTAL_MONSTERS	Общее кол-во монстров
LEVELINFO_KILLED_MONSTERS	Кол-во убитых монстров

### Количество монстров/итемов с определенных тэгом

```
int ThingCount(type, tid); // возвращает кол-во монстров типа type, имеющих тэг tid
```

Если type установлен в T\_NONE, функция подсчитает кол-во *всех* акторов с заданным тэгом (tid), невзирая на тип. Если tid установлен в 0, функция подсчитает кол-во *всех* акторов типа (type), невзирая на тэг.

Пример:

```
ThingCount(T_IMP, 5) // кол-во импов с тегом 5
```

```
ThingCount(T_IMP, 0) // общее кол-во всех импов на уровне
```

```
ThingCount(T_NONE, 9) // кол-во акторов с тэгом 9, невзирая на типы
```

## Актеры. Игроки.

### Общая информация об игроке

```
int GetPlayerInfo(int player_number, int player_info)
```

player\_number – номер игрока, по которому нужна информация

player\_info – тип запрашиваемой информации:

PLAYERINFO_TEAM	Команда игрока (255-без команды)
PLAYERINFO_AIMDIST	Дистанция автоприцеливания игрока
PLAYERINFO_COLOR	Цвет игрока (в формате 0xRRGGBB)
PLAYERINFO_GENDER	Пол игрока (0-мальчик, 1-девочка, 2-нечто другое)

*Пример: (делаем женский туалет недоступным лицам мужского пола :)*

```
script 1 (void)
{
    if(GetPlayerInfo(PlayerNumber(), PLAYERINFO_GENDER) == 1)
        Door_Open(1, 20);
    else
        Print(s:"sorry dude, ladies only");
}
```

### Свойства актора

- Получить/установить свойство

```
int GetActorProperty(int tid, int property)
SetActorProperty(int tid, int property, value)
```

tid – тэг актора. Используйте "0" для доступа к свойствам вызвавшего актора.

value – может быть типом int, float или str, в зависимости от *property*.



*Функция GetActorProperty не способна возвращать строковые property (из-за того, что ACS не умеет работать с динамическими строками). Используйте CheckActorProperty для получения строк (описана ниже).*

property:

APROP_Alpha	Значение alpha для стиля STYLE_Translucent
APROP_Ambush	Флаг засады AMBUSH
APROP_ChaseGoal	Цель юнита (актуально для монстров)
APROP_Damage	Урон ракеты актора
APROP_DamageFactor	damage factor актора
APROP_Dropped	Уроненный предмет (оружие, патроны, труп монстра). Уроненные предметы давятся закрывающимися дверями.
APROP_Friendly	Дружественный юнит (для монстров – враг)
APROP_Frightened	Монстр напуган (бежит от игрока)
APROP_Gravity	Фактор гравитации
APROP_Health	Уровень здоровья
APROP_Invulnerable	Юнит в состоянии неуязвимости
APROP_JumpZ	Высота прыжка игрока как fixed. Длину прыжка можно определить так: <i>Длина прыжка = (JumpZ*JumpZ)/2 + MaxStepHeight</i>
APROP_NoTarget	Актор не может быть целью для других монстров
APROP_RenderStyle	Стиль рендера актора: STYLE_None - не рисовать STYLE_Normal - обычный STYLE_Fuzzy - силуэт с эффектом "fuzz" STYLE_SoulTrans - как Lost Soul STYLE_OptFuzzy - пользовательские предпочтения STYLE_Translucent - полупрозрачно STYLE_Add - additive-блендинг
APROP_Score	Счетчик. Можно использовать для набранных очков.
APROP_Species	К какому виду принадлежит монстр
APROP_Speed	Скорость ракеты, фаерболла и т.п. Для игроков здесь всегда будет 1.0, а не действительная скорость!
APROP_ActiveSound	Звук юнита в состоянии ходьбы.
APROP_AttackSound	Звук юнита в состоянии атаки.
APROP_SeeSound	Звук юнита когда он заметил игрока.
APROP_PainSound	Звук юнита в состоянии боли.
APROP_DeathSound	Звук юнита в состоянии смерти.

*Пример: (проверка здоровья актора с тэгом tid, проверка стиля отрисовки актора с тэгом 12)*

```
script 1 (int tid)
{
```

```

if (GetActorProperty (tid, APROP_Health) < 25)
    print (s:"Thing ", d:tid, " has less than 25 health!!");
if (GetActorProperty (12, APROP_RenderStyle) == STYLE_OptFuzzy)
    print (s:"Thing 12 is probably a spectre!");
}

```

- Проверить значение свойства

```
bool CheckActorProperty(int tid, int property, int value)
```

Возвращает TRUE-если доступ к запрашиваемому свойству был успешен, FALSE-при неудаче.

tid – тэг актора (0-активатор скрипта);

property – свойство;

value – значение свойства, с которым нужно сравнить.

*Пример: (проверка класса актора и уничтожение его и всех его собратьев, если это был имп)*

```

script 1 (int tid)
{
    if (CheckActorProperty(tid, APROP_Species, "DoomImp"))
    {
        if (ThingCountName("DoomImp", tid) > 1)
            print(s:"These Imps must die!");
        else
            print(s:"This Imp must die!");s
            Thing_Destroy(tid, TRUE);
    }
}

```

### Свойства игрока

```
SetPlayerProperty(who, set, which)
```

who – 1 - для всех игроков, 0 - только для активировавшего скрипт игрока.

set – 1 - включить, 0 - выключить.

which:	
PROP_FROZEN (0)	Заморозить игрока (игрок может вертеться на месте, прыгать/приседать и стрелять)
PROP_TOTALLYFROZEN (4)	Полностью заморозить игрока (движение и любые действия недоступны)
PROP_NOTARGET (1)	Монстры не нападают на игрока
PROP_FLY (3)	Игрок может летать (гравитация не действует)
PROP_INVULNERABILITY (5)	Дает сферу неуязвимости. set=1 – обычный вариант, set=2 – без эффекта белого экрана
PROP_BUDDHA (16)	Режим Будды (1 неубавимый процент здоровья)

*Пример:*

```

SetPlayerProperty(1, 1, PROP_FROZEN); // заморозить игрока
SetPlayerProperty(1, 0, PROP_FROZEN); // разморозить игрока

```

*Хорошее применение этой функции – раздача уникальных тэгов игрокам в сетевой игре при старте уровня:*

```

script 5 ENTER
{
    Thing_ChangeTID(0, 1000 + PlayerNumber());
}

```

В результате сетевые игроки получают тэги 1000, 1001, 1002, ...

### Координаты

```

int GetActorX(tid)
int GetActorY(tid)
int GetActorZ(tid)

```

Возвращает координату в формате fixed.

```
bool SetActorPosition(int tid, fixed x, fixed y, fixed z, bool fog)
```

Устанавливает координаты x,y,z для актора tid, с тумана телепортера fog.

Возвращает TRUE при успехе, FALSE при неудаче.



*Координаты (x,y,z) НЕ являются теми же координатами, которые используются в редакторе!  
Для перевода координат редактора в fixed-координаты умножьте их на 65536.*

*Пример: (несем свечку над игроком)*

```

script 1 ENTER
{
    while (TRUE)
    {
        SetActorPosition(1, GetActorX(1), GetActorY(1), GetActorZ(0), 0);
        delay(1);
    }
}

```

```
}
}
```

### Уровень пола/потолка под/над актором

```
int GetActorFloorZ(int tid)
int GetActorCeilingZ(int tid)
```

### Скорости

```
int GetActorVelX(tid)
int GetActorVelY(tid)
int GetActorVelZ(tid)
```

Возвращают компоненты скорости (x,y,z) типа fixed.

```
bool SetActorVelocity(int tid, fixed vx, fixed vy, fixed vz, bool add, bool setbob)
```

tid – тэг объекта(ов), которому(ым) задается скорость; vx,vy,vz – компоненты скорости; add – TRUE-скорость прибавляется к уже имеющейся скорости объекта, FALSE-простое присвоение скорости; setbob – TRUE-установка скорости не затрагивает bobbing объекта, FALSE-затрагивает.

*Пример: (поместите этот код в скрипт OPEN своей карты и попрактикуйтесь в стрельбе по прыткому какедемону(-ам))*

```
int angle, pitch, velx, vely, velz;
while (GetActorProperty(tid, APROP_Health) > 0)
{
    angle = random(0, 1.0);
    pitch = random(-0.25, 0.25);
    velx = FixedMul(cos(angle), FixedMul(cos(pitch), 10.0));
    vely = FixedMul(sin(angle), FixedMul(cos(pitch), 10.0));
    velz = FixedMul(sin(pitch), 10.0);
    SetActorVelocity(tid, velx, vely, velz, FALSE, FALSE);
    delay(random(1, 7) * 5);
}
```

### Высота глаз актора

```
int GetActorViewHeight(int tid)
```

Возвращает высоту взгляда типа fixed.

### Угол взгляда актора (вертикальный)

```
int GetActorPitch(int tid)
SetActorPitch(int tid, int pitch)
```

Возвращает/устанавливает угол типа fixed в диапазоне [-0.25..0.25] (что соответствует [-90°..90°]).

Отрицательные значения соответствуют взгляду вверх.

Software-рендереры имеют ограничения по вертикальному углу, поэтому диапазон будет **уже**, чем [-0.25..0.25].

### Угол взгляда актора (горизонтальный (азимутальный))

```
int GetActorAngle(int tid)
SetActorAngle(int tid, int angle)
```

Возвращает/устанавливает угол типа fixed (0.25-север, 0.5-запад, 0.75-юг, 1.0-восток). Используйте деление на 256 (а именно >> 8) для получения байтового угла (64-север, 128-запад, 192-юг, 0-восток).

*Пример: (ускорение игрока в направлении его взгляда)*

```
script 10 ENTER
{
    ThrustThing(GetActorAngle(0) >> 8, 50, 1, 0);
}
```

### Классификация акторов ("чем является объект tid?")

```
int ClassifyActor(int tid)
```

tid – идентификатор объекта.

Возвращаемые значения:

ACTOR_NONE	Объект с тэгом tid не найден
ACTOR_WORLD	-
ACTOR_PLAYER	Игрок
ACTOR_BOT	Бот
ACTOR_VOODOODOLL	"вуду"
ACTOR_MONSTER	Монстр
ACTOR_ALIVE	Живой
ACTOR_DEAD	Мертвый
ACTOR_MISSILE	Ракета в полете
ACTOR_GENERIC	Другое (предмет интерьера, маркер, teleport destination и пр.)

## Принадлежность актора определенному классу

```
bool CheckActorClass(int tid, str class)
```

*Пример: (образно говоря – а такой ли ты имп, за которого себя выдаешь, юнит №1055 ?)*

```
if (!CheckActorClass(1055, "DoomImp"))
    print(s:"Unit 1055 is NOT an Imp.");
```

## Универсальная функция для определения класса актора

```
// массив Class должен быть глобальным
str Class[105] = {
"ShotgunGuy", "ChaingunGuy", "BaronOfHell", "ZombieMan", "DoomImp", "Arachnotron", "SpiderMastermind",
"Demon", "Spectre", "DoomImpBall", "Clip", "Shell", "Cacodemon", "Revenant", "Bridge", "ArmorBonus",
"Stimpack", "Medikit", "Soulsphere", "Shotgun", "Chaingun", "RocketLauncher", "PlasmaRifle", "BFG",
"Chainsaw", "SuperShotgun", "Rock1", "Rock2", "Rock3", "Dirt1", "Dirt2", "Dirt3", "Dirt4", "Dirt5",
"Dirt6", "PlasmaBall", "RevenantTracer", "SGShard1", "SGShard2", "SGShard3", "SGShard4", "SGShard5",
"SGShard6", "SGShard7", "SGShard8", "SGShard9", "SGShard0", "GreenArmor", "BlueArmor", "Cell",
"BlueCard", "RedCard", "YellowCard", "YellowSkull", "RedSkull", "BlueSkull", "ArchvileFire",
"StealthBaron", "StealthHellKnight", "StealthZombieMan", "StealthShotgunGuy", "LostSoul", "Archvile",
"Fatso", "HellKnight", "Cyberdemon", "PainElemental", "WolfensteinSS", "StealthArachnotron",
"StealthArchvile", "StealthCacodemon", "StealthChaingunGuy", "StealthDemon", "StealthDoomImp",
"StealthFatso", "StealthRevenant", "ExplosiveBarrel", "CacodemonBall", "Rocket", "BFGBall",
"ArachnotronPlasma", "Blood", "BulletPuff", "MegaspHERE", "InvulnerabilitySphere", "Berserk",
"BlurSphere", "RadSuit", "Allmap", "Infrared", "ClipBox", "RocketAmmo", "RocketBox", "CellPack",
"ShellBox", "Backpack", "Gibs", "ColonGibs", "SmallBloodPool", "BurningBarrel", "BrainStem",
"ScriptedMarine", "HealthBonus", "FatShot", "BaronBall"
};

function int GetActorClass (int tid)
{
    for (int i=0; i<105; i++)
        if (ThingCountName (Class[i], tid))
            return Class[i];
    return 0;
}
```

## Перерождение монстров/игроков

```
int MorphActor (int tid, [str playerclass, [str monsterclass, [int duration, [int style, [str
morphflash, [str unmorphflash]]]]])
int UnMorphActor (int tid[, bool force])
```

Подробнее:

<http://zdoom.org/wiki/MorphActor>

<http://zdoom.org/wiki/UnMorphActor>

## Смена уровня

```
ChangeLevel(str map_name, int position, int flags [,int skill])
```

Более мощный аналог того эффекта, который достигается при использовании IDCLEV##.

ChangeLevel переходит на другой уровень в определенную точку, а также опционально – меняет сложность игры. map\_name – lump-имя карты, position – номер стартовой позиции игрока ("Player 1 start", "Player 2 start", ...).

flags:	Что это	Рекомендации (для чего хорошо подойдет)
CHANGELEVEL_KEEPPACING	Сохранить азимутальный угол (направление взгляда) игрока	Для уровней типа "hub"
CHANGELEVEL_NOINTERMISSION	Не показывать заставку	Для уровней типа "hub"
CHANGELEVEL_NOMONSTERS	Играть без монстров	Повторный вход в уровень, но по сюжету не связанный с битвами
CHANGELEVEL_RESETHALTH	Восстановить здоровье до 100	Для уровней с боссами / трудных уровней
CHANGELEVEL_RESEFINVENTORY	Очистка инвентаря	Например, для выравнивания баланса оружия (если не удалось сделать это грамотной расстановкой боеприпасов на уровне)

skill – новый уровень сложности. Хорошо подходит для создания skill-селекторов, как в первом Quake.



*Если не указать skill, уровень будет запущен на сложности "0" (SKILL\_VERY\_EASY). Для сохранения выбранного уровня сложности используйте "-1".*

*Пример: (смена уровня на E3M1 на сложности hard, с сохранением направления взгляда игрока и очистка инвентаря)*

```
script 1 (void)
{
```

```
ChangeLevel ("E3M1", 0, CHANGLELEVEL_RESEFINVENTORY|CHANGELEVEL_KEEPPACING, SKILL_HARD);  
}
```

## Мультиплеер

### Количество подключившихся игроков

```
int PlayerCount()
```

Для синглплеерной игры всегда возвращает 1.

*Пример – порождение кибердемона при большом числе игроков на уровне:*

```
script 12 (void)
{
    if (PlayerCount() >= 6)
        thing_spawn(1, T_CYBERDEMON, 0, 0);
}
```

*Пример – ожидание всех игроков в определенной зоне уровня:*

```
int count = 0;

// Use for Actor enters sector
script 10 (void)
{ count++; }

// Use for Actor leaves sector
script 11 (void)
{ count--; }

script 100 OPEN
{
    while (count < PlayerCount())
        Delay(35);
    PrintBold(s:"All players ready!");
    ...
}
```

### Номер игрока

```
int PlayerNumber()
```

Возвращает номер игрока вызвавшего скрипт, начиная с 0. Если скрипт вызван не игроком, возвращает -1.

*Пример – присвоение уникального тэга каждому игроку, зашедшему на уровень: (1000, 1001, 1002, ...)*

```
script 5 ENTER
{
    Thing_ChangeTID(0, 1000 + PlayerNumber());
}
```

### Проверка участия игрока

```
bool PlayerInGame(int player_number)
```

player\_number – номер игрока [0..7].

### Фраги игрока

```
int PlayerFrag()
```

Возвращаемое число может быть и отрицательным, если игрок покончил с собой. Если скрипт вызван НЕ игроком, всегда возвращает 0.

### Проверка на бота

```
bool PlayerIsBot(int player_number)
```

Возвращает TRUE, если игрок [0..7] является ботом.

*Пример: (отдача элемента паззла или умения боту вместо игрока может сделать игру непроходимой. Приведенный скрипт предотвращает это)*

```
script 55 (void)
{
    int marine = Random(0, PlayerCount());
    while (PlayerIsBot(marine))
        marine = Random(0, PlayerCount()); // Pick another marine

    SetActorProperty(1000 + marine, APROP_INVULNERABLE, TRUE);
    PrintBold(n:marine+1, s:" is totally invulnerable!");
}

script 1000 ENTER
{
    Thing_ChangeTID(0, 1000+PlayerNumber()); //присвоение уникальных тэгов игрокам (и ботам)
}
```

## Оружие / Инвентарь

### Смена оружия

```
bool SetWeapon(str weapon_item)
```

Вернет TRUE если оружие успешно сменено, и FALSE в обратном случае.

### Емкость боеприпасов игрока

```
int GetAmmoCapacity(str class_name)
```

Значения по умолчанию (без backpack / с backpack):

Clip	200/400
Shell	50/100
RocketAmmo	50/100
Cell	300/600

### Установить емкость боеприпасов

```
int SetAmmoCapacity(str class_name, int max_amount)
```

class\_name – допустимые значения: Clip, Shell, RocketAmmo, Cell.

### Проверка инвентаря игрока/актеров

```
int CheckActorInventory(int tid, str inventory_item)
```

Возвращает кол-во предметов инвентаря *inventory\_item* у актора с тэгом *tid*.

inventory\_item:

Оружие	Боеприпасы	Ключи
Fist	Clip (10)	BlueCard
Chainsaw	Shell (4)	BlueSkull
Pistol	RocketAmmo (1)	RedCard
Shotgun	Cell (20)	RedSkull
SuperShotgun		YellowCard
Chaingun	Рюкзак	YellowSkull
RocketLauncher	Backpack	
PlasmaRifle		Авто-карта
BFG9000		Allmap

Если в своей карте вы часто работаете с оружием/инвентарем, вам могут пригодиться эти глобальные массивы:

```
str weapons[8] = {"Pistol", "Shotgun", "SuperShotgun", "Chaingun", "RocketLauncher",
                 "PlasmaRifle", "BFG9000", "Chainsaw"};
str ammo[4] = {"Clip", "Shell", "RocketAmmo", "Cell"}
str keys[6] = {"BlueCard", "BlueSkull", "RedCard", "RedSkull", "YellowCard", "YellowSkull"}
```



Функция не рассматривает *tid=0* как игрока!  
Для проверки инвентаря игрока вызывайте *CheckInventory*.

- Кол-во определенного предмета инвентаря у игрока

```
int CheckInventory(str inventory_item)
```

Возвращает кол-во предметов инвентаря *inventory\_item* у игрока.

Пример – проверяем наличие у игрока дробовика и минимум 21-го патрона к нему:

```
script 52 (void)
{
    if (CheckInventory("Shotgun") && CheckInventory("Shell") > 20)
        Print(s:"Use the shotgun to take out those 20 imps!");
    else
        Print(s:"Run away from the imps you loser!");
}
```



**(информация неточная, необходима проверка)**

Этой функцией нельзя определить наличие у игрока: *Partial Invisibility*, *Light Amplification Visor*, *Radiation Suit*, *Invulnerability*. Для проверки этих паверапов проверяйте встроенную переменную *Powerup*: *BlurSphere* (=Partial Invisibility), *Infrared* (=Light Amplification Visor), *RadSuit* (=Radiation Suit), *InvulnerabilitySphere* (=Invulnerability).

### Очистка инвентаря

- Очистка инвентаря игрока

```
ClearInventory(int tid)
```

Функция полностью очищает инвентарь актора с тэгом *tid*.

*Пример: (отбираем всё оружие у игрока)*

```
script 50 (void)
{
    Print(s:"You hand your weapons over to security.");
    ClearInventory();
    GiveInventory("Fist", 1);
}
```

#### • Очистка инвентаря актора

```
ClearActorInventory(int tid)
```

Функция полностью очищает инвентарь актора с тэгом *tid*.



*Функция не действует на предметы инвентаря с флагом INVENTORY.UNDROPPABLE. Для принудительного отбирания таких предметов используйте TakeInventory.*

*Пример: (в deathmatch каждые 5 минут отбираем все оружие у случайного игрока)*

```
script 1 enter
{
    Thing_ChangeTID(0, 1000 + PlayerNumber());
}

script 2 open
{
    int p;
    while (TRUE)
    {
        delay(35 * 60 * 5);
        do {
            p = random(0, 7);
        } while (!PlayerInGame(p));

        ClearActorInventory(1000 + p);
        HUDMessageBold(n:p+1, s:" is begging for a quick death!";
            HUDMSG_FADEOUT, 1, CR_RED, 0.5, 0.5, 3.0, 1.0);
    }
}
```

#### Отбор предмета инвентаря

```
TakeInventory(str inventory_item, int amount)
TakeActorInventory(int tid, str inventory_item, int amount)
```

*inventory\_item* – тип отбираемого предмета инвентаря, *amount* – количество.

*tid* – тэг актора.

*Пример:*

```
TakeInventory("SuperShotgun", 1); // отбираем двустволку
TakeInventory("ShellBox", 5); // отбираем 5 патронов для дробовика
```

#### Выдача предмета инвентаря

```
GiveInventory(str inventory_item, int amount)
GiveActorInventory(int tid, str inventory_item, int amount)
```

*Пример: (даем игроку ракетланчер)*

```
GiveInventory("RocketLauncher", 1);
```

*Пример: (выдаем дробовик каждому игроку, у которого его не было)*

```
script 1 enter
{
    Thing_ChangeTID(0, 1000 + PlayerNumber());
}

script 2 (void)
{
    for (int p = 0; p < 8; p++)
        if (PlayerInGame(p) && !CheckActorInventory(1000 + p, "Shotgun"))
            GiveActorInventory(1000 + p, "Shotgun", 1);
}
```

#### Вооружение NPC

Маринесы рождаются, имея лишь пистолет (как игрок). Чтобы дать маринесу плазмаган, пишем такой код:

```
SetMarineWeapon(10, MARINEWEAPON_PlasmaRifle);
```



## Порождение снаряда

```
SpawnProjectile(int tid, string type, int angle, int speed, int vspeed, int gravity, int newtid)
```

tid – объект, производящий выстрел;

type – тип снаряда: "ArachnotronPlasma", "ArchvileFire", "BaronBall", "CacoDemonBall", "BFGBall", "BFGExtra", "BulletPuff", "DoomImpBall", "FatShot" (fireball манкубуса), "PlasmaBall", "RevenantTracer", "RevenantTracerSmoke" (самонаводящийся снаряд), "Rocket".

angle – байтовый угол; speed – горизонтальная скорость снаряда; vspeed – вертикальная скорость снаряда;

gravity – гравитация, действующая на снаряд (заметьте, по умолчанию на снаряды монстров действует нулевая грав.);

newtid – новый tid, присваиваемый созданному снаряду.

*Пример: (выстрел объектом с tid=1 случайного снаряда по случайному направлению, с горизонтальной скоростью 20)*

```
str projectile[3] = {"DoomImpBall", "CacodemonBall", "BaronBall"};
SpawnProjectile(1, projectile[random(0, 2)], random(0, 255), 20, 0, 0, 0);
```

## Телепортация

### Телепортация объекта

```
TeleportOther(tid, destination_tid, fog)
```

Телепортация предмета *tid* в MapSpot с тэгом *destination\_tid*, с туманом (fog=1) или без (fog=0).

### Телепортация активатора (объекта, вызвавшего скрипт)

```
Teleport(destination_tid, tag, nofog)
```

Телепортация активатора в MapSpot с тэгом *destination\_tid* в сектор с тэгом *tag*, со вспышкой в точке отправки (nofog=0) или без (nofog=1).

Если tag=0, точка назначения будет выбрана случайно.

## Освещение

### Освещенность сектора

```
Light_ChangeToValue(tag, value)
```

### Цвет освещения сектора

```
Sector_SetColor(tag, r, g, b, desat)
```

tag – тэг сектора

r,g,b – новый цвет освещения

desat – параметр в диапазоне [0..255], определяет силу обесцвечивания сектора (255 – полное обесцвеч.)



*Почитатели олдскульной атмосферы неприязненно относятся к цветному освещению (в GZDoom / JDoom в первую очередь). Используйте на свой страх и риск.*

### Цвет тумана сектора

```
Sector_SetFade(tag, r, g, b)
```

Устанавливает цвет, которым будет оттеняться сектор при удалении от него игрока (эффект тумана).

tag – тэг сектора

r,g,b – цвет "тумана"

### Освещенность актора/игрока

```
int GetActorLightLevel(int tid)
```

*Пример: (делаем босса полупрозрачным, если его освещенность меньше 40)*

```
script 1 (int boss)
{
    if (GetActorLightLevel(0) < 40)
        SetActorProperty (boss, APROP_RENDERSTYLE, STYLE_Fuzzy);
}
```

## Экранные пост-спецефффекты

### Окрашивание экрана

```
FadeTo(int red, int green, int blue, fixed amount, fixed seconds);
```

(red, green, blue) – цвет

amount – должен быть в пределах [0.0..1.0]

seconds – длительность цветового перехода (секунд).

*Пример:*

```
script 100 enter
{
    //Полная краснота в глазах через 3 секунды
    fadeto (255, 0, 0, 1.0, 2.0);
    delay(35 * 2);

    //Половинная чернота в глазах через 2 секунды
    fadeto (0, 0, 0, 0.5, 2.0);
}
```

### Плавная смена оттенка экрана

```
FadeRange(int red1, int green1, int blue1, fixed amount1,
           int red2, int green2, int blue2, fixed amount2, int seconds)
```

Функция сразу выставляет оттенокок (*red1, green1, blue1*), и создает плавный цветовой переход к (*red2, green2, blue2*) в течение времени *seconds*.

*Пример: (цветовой переход от 80% зеленого к 80% синего в течение 2 секунд)*

```
faderange (0, 255, 0, 0.8, 0, 0, 255, 0.8, 2.0);
```

### Смена палитры объектов "на лету"

Этими функциями можно легко сделать кибердемона кислотных тонов или что-то в этом духе:

```
CreateTranslation(int transnumber, a:b=[red1,green1,blue1]:[red2,green2,blue2], ...)
Thing_SetTranslation (tid, translation)
```

Подробнее:

<http://zdoom.org/wiki/CreateTranslation>

[http://zdoom.org/wiki/Thing\\_SetTranslation](http://zdoom.org/wiki/Thing_SetTranslation)

## Физика

### Гравитация (сила тяжести)

```
SetGravity(fixed amount)
```

amount: 800 – нормальная гравитация, 400.0 – пониженная вдвое, 1600.0 – увеличенная вдвое и т.д.

### Сопrotивление воздуха

```
SetAirControl(int amount)
```

Сопrotивление воздуха действует *только* когда игрок находится не на земле (допустим, прыгнул), и не в воде. Значение по умолчанию 0.00390625.



*Сопrotивление воздуха – не такой уж безобидный параметр, как кажется на первый взгляд. Изменение значения по умолчанию значительно увеличивает (или уменьшает) расстояние, которое игрок может пролететь при прыжках с краев зданий, обрывов и т.д., и в конечном счете может радикально изменить проходимость вашего уровня.*

### Приложение силы к предмету

```
ThrustThing(angle, force, nolimit, tid)
```

angle – байтовый угол; для получения направлений "вправо", "назад", "влево" – добавляйте к углу +64, +128, +192; force – сила, в юнитах/сек (1 сек = 35 тиков); nolimit – должно быть 1, если force > 30; *(не пойму, зачем вообще нужен этот параметр – примеч. VladGuardian)* tid – тэг предмета, к которому надо приложить тягу.

Пример:

```
ThrustThing(angle*256/360, 0, 0, 0)
ThrustThing(angle*256/360+64, 0, 0, 0)
```

### Приложение вертикальной силы к предмету

```
ThrustThingZ(tid, force, up_down, set_add)
```

tid – тэг предмета, к которому надо приложить тягу; force – сила, в юнитах/сек, деленная на 4 (1 сек = 35 тиков); up\_down – направление (0-вверх, 1-вниз); set\_add – (0-обнуляет вертикальную скорость предмета, затем прикладывает силу, 1-просто прикладывает силу); ThrustThingZ удобно комбинировать с ThrustThing.

### Землетрясение

```
Radius_Quake2(tid, intensity, duration, damrad, tremrad, str sound)
```

intensity – интенсивность землетрясения [1..9]; duration – длительность в тиках; damrad – радиус повреждений в 64x64; tremrad – радиус тряски в 64x64; tid – тэг предмета-эпицентра землетрясения; sound – звук землетрясения.

*Пример: (землетрясение в течение 5 сек вокруг объекта tid=10 радиусом 2000, радиусом повреждений 1000, без звука)*

```
script 1 OPEN
{
    Radius_Quake2(10, 3, 5*35, 1000, 2000, "");
}
```

## Под водой

### Запас воздуха в легких игрока

```
int GetAirSupply(int player_num)
```

Результат - в тиках. Если значение отрицательное – значит игрок уже тонет (у игрока отнимается здоровье).

```
bool SetAirSupply(int player_num, int tics)
```

Устанавливает запас воздуха в тиках. Функция вернет 1 в случае успеха, и 0 если игрока *player\_num* не существует. Функция действует только на игроков, т.к. у монстров нет такой величины.

Пример:

```
script 1 ENTER
{
    while (1) {
        print(s:"You will start to drown in ", i:GetAirSupply(0)/35, s:" sec");
        delay(1);
    }
}
```

## Модификация уровня

### Двери

```
Door_Open(tag, speed, light_tag);
Door_Close(tag, speed, light_tag);
```

tag – тэг сектора, представляющего дверь; speed – скорость открытия;  
light\_tag – тэг сектора, где будет выполнен плавный световой переход. Сектор имеет освещенность *самого темного* ближайшего сектора, когда дверь закрыта, и *самого светлого* ближайшего – когда будет полностью открыта.

### Задать повреждение для сектора

```
Sector_SetDamage(tag, amount, mod)
```

tag – тэг сектора;  
amount – сила вреда:

< 20	Сектор не вредит игроку, одетому в environment suit
20 - 49	Сектор иногда вредит игроку, одетому в environment suit
> 50	Сектор всегда вредит игроку, одетому в environment suit, но не действует на игрока с Invulnerability.

mod:

0 = MOD_UNKNOWN	16 = MOD_TELEFRAG	20 = MOD_EXIT	25 = MOD_DISINTEGRATE (ala Strife's Mauler)
12 = MOD_WATER	17 = MOD_FALLING	21 = MOD_SPLASH	26 = MOD_POISON
13 = MOD_SLIME	18 = MOD_SUICIDE	22 = MOD_HIT	27 = MOD_ELECTRIC
14 = MOD_LAVA	19 = MOD_BARREL	23 = MOD_RAILGUN	
15 = MOD_CRUSH	1000 = Massacre (no const)	24 = MOD_ICE (ala Hexen)	

### Повредить всем объектам в секторе

```
SectorDamage(int tag, int amount, str type, str protection_item, int flags)
```

tag – тэг сектора;  
Причиняет вред, только когда вызвана функция. Сам сектор остается "безвредным".  
type – тип вреда (см. [http://zdoom.org/wiki/Damage\\_types](http://zdoom.org/wiki/Damage_types) )  
protection\_item – оставить 0 для Doom (актуально только для Heretic/Hexen).  
flags: (можно объединять несколько флагов через символ | )

DAMAGE_PLAYERS	Вред всем игрокам в этом секторе.
DAMAGE_NONPLAYERS	Вред всем не-игрокам в этом секторе (nonplayers – те, которых все же можно подстрелить).
DAMAGE_IN_AIR	Вред даже тем, кто в воздухе или под водой.
DAMAGE_SUBCLASSES_PROTECT	Нет вреда тем, кто несет итем, унаследованный от <i>protection_item</i> (или кто сам является этим итемом)

Пример: (убийство любого не-игрока, вошедшего в сектор. "Assign it to run from an **Actor Enters Sector** thing placed in the sector with the matching tag" – не смог толком перевести – примеч. VladGuardian)

```
script 1 (int tag)
{
    if (PlayerNumber() < 0)
    {
        printbold(s:"Kill the non-players!!!");
        Sector_SetFade(tag, 255, 0, 0);
        while (GetActorProperty(0, APROP_Health) > 0)
        {
            SectorDamage(tag, 100, "Fire", 0, DAMAGE_NONPLAYERS | DAMAGE_IN_AIR);
            delay(5);
        }
        Sector_SetFade(tag, 0, 0, 0);
    }
}
```

### Повредить активатору скрипта

```
DamageThing(amount, mod)
```

amount – сумма ущерба :). Отрицательное значение – лечение. Нулевое значение – гарантированное убийство активатора.  
mod – сообщение, появляющееся после смерти игрока (obituary message).

### Замена текстур глобальная

```
ReplaceTextures(str old_texture_name, str new_texture_name [, int flags])
```

Заменяет в уровне все текстуры *old\_texture\_name* на *new\_texture\_name*.  
flags – необязательный параметр, указывает, какие текстуры НЕ надо менять:

NOT_BOTTOM	Не менять нижние текстуры стен.
NOT_MIDDLE	Не менять средние текстуры стен.
NOT_TOP	Не менять верхние текстуры стен.
NOT_FLOOR	Не менять текстуры пола.
NOT_CEILING	Не менять текстуры потолка.



Функцию хорошо применять для создания пугающей атмосферы, например, при нажатии свитча все вокруг вдруг преобразуется в мрачные тона или кровавые текстуры.

### Замена текстур полов/потолков

```
ChangeFloor(int tag, str flat_name)
ChangeCeiling(int tag, str flat_name)
```

Меняет текстуру полов/потолков для всех секторов с тэгом *tag*, на *flat\_name*.

### Замена текстур стены (лайндефа)

```
SetLineTexture(int line_id, int line_side, int sidedef_texture, str texture_name)
```

*line\_id* – идентификатор линии (стены);

*line\_side* – сторона стены (SIDE\_FRONT-передняя, SIDE\_BACK-задняя);

*sidedef\_texture* – секция стены: TEXTURE\_TOP, TEXTURE\_MIDDLE или TEXTURE\_BOTTOM;

*texture\_name* – имя текстуры.

*Пример:* (убрать среднюю текстуру на передней части *line1*, наложить текстуру *BFALL1* на задней части *line2*)

```
script 1 (int line1, int line2)
{
    SetLineTexture(line1, SIDE_FRONT, TEXTURE_MIDDLE, "-"); //remove middle
    SetLineTexture(line1, SIDE_FRONT, TEXTURE_MIDDLE, "-"); //floating texture
    SetLineTexture(line2, SIDE_BACK, TEXTURE_TOP, "BFALL1");
}
```

### Небо

- Изменение текстуры неба

```
ChangeSky(str sky1, str sky2)
```

*sky1* – первая текстура неба, *sky2* – вторая.

*sky1* должен совпадать с *sky2*, если *double sky* включен (в *MAPINFO*).

*Пример:* (анимированная вода – в качестве неба)

```
ChangeSky("FWATER1", "FWATER1");
```

- Изменение скорости скроллинга текстуры неба

```
SetSkyScrollSpeed(int sky, float sky_speed)
```

*sky* – 1 либо 2, скорость скроллинга.

### Зеркальный пол

```
Sector_SetPlaneReflection(tag, floor, ceiling)
```

*tag* – тэг обрабатываемого сектора;

*floor* – степень зеркальности пола [0..255];

*ceiling* – степень зеркальности потолка [0..255].

*Пример:* (делаем зеркальными – пол сектора с тэгом 1 и потолок сектора с тэгом 2)

```
script 1 OPEN
{
    Sector_SetPlaneReflection(1,128,0);
    Sector_SetPlaneReflection(2,0,128);
}
```

### Блокирование проходов (лайндефов)

```
SetLineBlocking(int line_id, int setting)
```

*setting*:

BLOCK_NOTHING	Снимает любую блокировку с линии
BLOCK_CREATURES	Блокирует игроков и монстров
BLOCK_EVERYTHING	Блокирует всё
BLOCK_PLAYERS	Блокирует только игроков (не монстров)



Особой необходимости в функции *SetLineBlocking* нет, в редакторе все это доступно в свойствах лайндефа:



### Блокирование прохода только для монстров

```
SetLineMonsterBlocking(int line_id, int setting)
```

setting – 1-монстры не пройдут, 0-снятие блокировки.

*Пример: (установка блокировки на линиях с тэгом 1 на 5 минут, снятие блокировки по прошествии 5 минут)*

```
script 1 open
{
    SetLineMonsterBlocking(1, 1);
    delay(35 * 60 * 5);
    SetLineMonsterBlocking(1, 0);
    HUDMessageBold(s:"ZOMG! The monsters can break through now!!!";
        HUDMSG_FADEOUT, 1, CR_RED, 0.5, 0.5, 3.0, 1.0);
}
```

### Привязка события к лайндефу

```
SetLineSpecial(int line_id, int special [, int arg0 [, int arg1 [, int arg2 [, int arg3 [, int arg4]]]])
```

*Пример: (установка выполнения скрипта 10 на лайндеф 1)*

```
SetLineSpecial(1, ACS_Execute, 10);
```

*Пример: (все линии с id=9 будут выполнять скрипт 10)*

```
script 1 (void)
{
    print (s:"setting action");
    ActivatorSound("misc/chat", 127);
    SetLineSpecial (9, ACS_Execute, 10);
}
script 10 (void)
{
    ActivatorSound("misc/chat", 127);
}
```

### Привязка события к сектору (на подъем/опускание сектора)

```
SetFloorTrigger (int tag, int height, int special [, int arg1 [, int arg2 [, int arg3 [, int arg4 [, int arg5]]]])
```

```
SetCeilingTrigger (int tag, int height, int special [, int arg1 [, int arg2 [, int arg3 [, int arg4 [, int arg5]]]])
```

При подъеме/опускании пола/потолка сектора *tag* на указанную разницу высот *height* – срабатывание события *special* с параметрами: *arg1, arg2, arg3, arg4, arg5*.



*height* – не высота, а *разница высот* (128 – поднятие на 128 map units, -64 – опускание на 64 map units)

*Пример: (выполнение скрипта 18 при поднятии потолка на 128)*

```
SetCeilingTrigger (12, 128, acs_execute, 18, 0);
```

*Пример: (телепортация активатора в tarspot #15 при опускании потолка на 64)*

```
SetCeilingTrigger (25, -64, teleport, 15);
```

## Звук и музыка

Есть 3 категории звуков в ZDoom:

- 1) музыка
- 2) окружающий звук (ambient sound) – звук, слышимый одинаково по всему уровню
- 3) точечный звук (sector sound) – звук, издаваемый одной точкой уровня, затухает с расстоянием

### Музыка

Поддерживается множество форматов музыки: MID, MUS (Doom music), MP3, OGG, WAV, FLAC, WMA, AIFF, MOD, ST3, ...

- Запустить встроенный в doom2.wad трек D\_RUNNIN:

```
SetMusic("D_RUNNIN");
```

- Запустить mp3-трек Metallica-"Master Of Puppets", предварительно загруженный в wad-редакторе:

```
SetMusic("MAST_PUP");
```

- Сбросить на музыку по умолчанию (для этого уровня):

```
SetMusic("");
```

Имеется также вариант функции SetMusic, срабатывающий *только* для игрока, активировавшего скрипт:

- Запуск музыки для определенного игрока:

```
LocalSetMusic(str song, opt int pattern);
```

### Окружающий звук

- Проиграть звук "handel/messiah" на полной громкости:

```
AmbientSound("handel/messiah", 127); // 0..127 volume
```

### Точечный звук

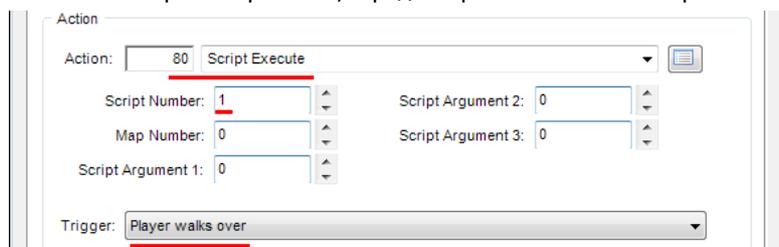
Наконец, самый *интересный* тип звука в ZDoom, позволяющий заметно усилить атмосферу вашего уровня.

```
SectorSound(str sound, int volume);
```

*Пример: (когда игрок пересечет linedef, на который навешен этот скрипт, половица скрипнет!)*

```
script 1 (void)
{
    SectorSound("world/creak1", 127);
}
```

Чтобы этот скрипт заработал, в редакторе "навешиваем" скрипт на linedef *нужного нам* сектора:



- Проиграть звук в позиции предмета:

```
ThingSound(int tid, str sound, int volume);
```

*Пример: (звук выключения генератора)*

```
// Предмет-источник звука (tid=1) должен быть близко к самому генератору (или совпадать)
ThingSound(1, "ambient/poweroff", 127);
```

- Проиграть звук от имени активатора:

```
ActivatorSound(str sound, int volume);
```

*Пример:*

```
script 4 (void) {
    ActivatorSound("misc/i_pkup", 127);
    print(s:"Sounds like you got something!");
}
```

Полный список стандартных звуков Doom:

[http://zdoom.org/wiki/Predefined\\_Sounds](http://zdoom.org/wiki/Predefined_Sounds)

Полный список команд, связанных с музыкой и звуком:

[http://zdoom.org/wiki/Built-in\\_ACS\\_functions#Sounds](http://zdoom.org/wiki/Built-in_ACS_functions#Sounds)

## Камеры

### Через какую камеру смотрит игрок?

```
int CheckPlayerCamera(int player)
```

player – номер игрока.

*Пример: (скрипт деактивирует акторов с тэгом=60, пока игрок смотрит через камеру, т.е. враги его не атакуют)*

```
script 10 ENTER
{
    while (!CheckPlayerCamera(PlayerNumber())) delay(1);
    Thing_Deactivate(60);
    while (CheckPlayerCamera(PlayerNumber())) delay(1);
    Thing_Activate(60);
    Restart;
}
```

### Отображение вида из камеры на текстуру

```
SetCameraToTexture(int camera_id, str texture_name, int fov)
```

Функция привязывает именованную текстуру *texture\_name* к камере *camera\_id*. Всё, что видит камера в поле своего зрения, будет сразу отображаться на текстуре.

camera\_tid – идентификатор камеры;

texture\_name – имя используемой текстуры. Текстура обязательно должна быть прописана в lump ANIMDEFS;

fov – поле зрения камеры (угол). (fov="field of view")

"Отвязать" текстуру от камеры нельзя, но можно привязать ее к другой камере. Вид с камеры будет обновляться только тогда, когда эта текстура будет видна игроком. Это естественная закономерность, что значительно экономит ресурсы системы (как CPU, так и GPU). Рендерить один вид из камеры не слишком затратно для системы. Ситуация несколько осложняется, когда игрок одновременно видит несколько видов из разных камер.

Автором этого учебника были проведены тестовые замеры и установлена следующая зависимость (на системе):

*(таблица будет посчитана в следующей версии учебника – примеч. VladGuardian)*

Конфигурация компьютера	Кол-во камер	fps	Конфигурация компьютера	Кол-во камер	fps
Одноядерные CPU					
ПК:	1				
Pentium-4 2GHz	2				
GeForce 7300 256Mb	3				
RAM 768Mb	4				
WinXP SP3	5				
Двухядерные CPU					
ПК:	1		Ноутбук:	1	
Core i3-540M	2		Core i5-2430M	2	
GeForce 220M	3		GeForce 520M	3	
RAM 2Gb	4		RAM 4Gb	4	
Win7	5		Win7 64-bit	5	

*Пример:*

ANIMDEFS:

```
Camertexture TCAMTEX1 128 64 fit 80 56
```

*Запуск уровня:*

```
script 1 OPEN
{
    SetCameraToTexture( 7, "TCAMTEX1", 90);
}
```

*Скрипт, переключающий вид на другую камеру:*

```
script 2 (int camera_tid, int fov)
{
    SetCameraToTexture( 7, "TCAMTEX1", fov);
}
```

*Реализация автоматически переключающихся видов из камер (через равные промежутки времени):*

```
script 3 OPEN
{
    int inter_delay = 70; // 2 seconds between switching
    while( 1 ) {
        SetCameraToTexture( 7, "TCAMTEX1", 90 );
        delay( inter_delay );
        SetCameraToTexture( 12, "TCAMTEX1", 90 );
        delay( inter_delay );
        SetCameraToTexture( 15, "TCAMTEX1", 40 ); // this one's zoomed in!
        delay( inter_delay );
    }
}
```

## Консоль

### Логирование (вывод строк в консоль)

Когда вы ведете разработку своего уровня, активно пишете скрипты, то как правило, что-то работает не так (а может быть, и сразу всё работает не так ☺ («Хьюстон, у нас проблемы!»)) Для этого существует функция логирования, позволяющая отображать строки и любые переменные прямо в консоль, для отслеживания их значений. Логирование можно рассматривать как разновидность print-a, но печатающего не на экран, а в консоль:

```
Log(s:"Level 2 loaded ok.");  
Log(s:"num_monsters = ", i:num_monsters);
```

Преимущество консоли перед экраном очевидно – сообщения с экрана довольно быстро исчезают, а в консоли сохраняются до конца работы.

### Получение значений переменных консоли

Полезным может также оказаться обращение к переменным консоли:

```
int GetCVar (str cvar)
```

Пример:

```
int screenblocks = GetCVar ("screenblocks");
```

## Заключение

That's all, folks!



# Приложение

## Типы предметов

<i>Точки входа</i>	<i>Обычные монстры</i>	<i>Оружие</i>	<i>Препятствия</i>	<i>Источники света</i>
Игрок 1 (1) Игрок 2 (2) Игрок 3 (3) Игрок 4 (4) Игрок 5 (4001) Игрок 6 (4002) Игрок 7 (4003) Игрок 8 (4004) Deathmatch (11)	Arachnotron (68) Archvile (64) Baron of Hell (3003) Cacodemon (3005) Chaingun Guy (65) Commander Keen (72) Cyberdemon (16) Demon (3002) Zombieman (3004) Shotgun guy (9) Hell Knight (69) The Icon of Sin (88) Imp (3001) Lost Soul (3006) Mancubus (67) Monster Cube Spawner (89) Monster Cube Target (87) Pain Elemental (71) Revenant (66) Spectre (58) Spider Mastermind (7) Wolfenstein Nazi (84)	BFG 9000 (2006) Chaingun (2002) Chainsaw (2005) Plasma Rifle (2004) Rocket Launcher (2003) Shotgun (2001) Super Shotgun (82)	Exploding Barrel (2035) Burning Barrel (70) Evil Eye (41) Floating skull rock (42) Grey tree (43) Large brown tree (54) Short green pillar (31) Short green pillar w/beatng heart (36) Short red pillar (33) Short red pillar w/skull (37) Stalagmite (47) Tall green pillar (30) Tall red pillar (32) Tall tech. pillar (48)	Candelabra (35) Candle (34) Floor lamp (2028) Short blue firestick (55) Short green firestick (56) Short red firestick (57) Short techno floor lamp (86) Tall blue firestick (44) Tall green firestick (45) Tall red firestick (46) Tall techno floor lamp (85)
<b>Объекты редактора</b>		<b>Боеприпасы</b>		<b>Динамический свет</b>
Visual Mode Camera(32000)		Ammo Clip (2007) Backpack (8) Box of Ammo (2048) Box of Rockets (2046) Box of Shells (2049) Cell Charge (2047) Cell Charge Pack (17) Rocket (2010) 4 Shotgun Shells (2008)		Flickering (9802) Flickering Additive (9812) Flickering Subtractive(9822) Pulsing (9801) Pulsing Additive (9811) Pulsing Subtractive (9821) Random Flickering (9804) Random Flickering Additive (9814) Random Flickering Subtractive (9824) Sector Static (9803) Sector Static Additive(9813) Sector Static Subtractive (9823) Static (9800) Static Additive (9810) Static Subtractive (9820)
<b>Камеры и интерполяция</b>		<b>Здоровье</b>	<b>Прочие декорации</b>	
Actor Mover (9074) Aiming Camera (9073) Camera (9025) Interpolation point (9070) Interpolation Special (9075) Moving Camera (9072) Path Follower (9071) Patrol Special (9047) Skybox Picker (9081) Skybox Viewpoint (9080) Security Camera (9025)		Health Bonus (2014) Stimpack (2011) Medikit (2012)	5 skulls on a pole (28) Bloody mess 1 (10) Bloody mess 2 (12) Dead cacodemon (22) Dead demon (21) Dead zombieman (18) Dead shotgun guy (19) Dead imp (20) Dead lost soul (23) Dead Marine (15) Hanging Leg (62) Hanging Leg (solid) (53) Hanging pair of legs (60) Hanging pair of legs (непроходимый) (52) Hanging torso (с удаленным мозгом) (78) Hanging torso (смотрит вниз) (75) Hanging torso (смотрит вверх) (77) Hanging torso (Open Skull) (76) Hanged one legged person (61) Hanged one legged person (непроходимый) (51) Hanging person with arms out (59) Hanging person with arms out (непроходимый) (50) Hanging person without brain of guts (74) Hanging person without guts (73) Twitching hanging person (63) Twitching hanging person (непроходимый) (49) Impaled Human (25) Imploded Human (Twitching) (26) Pile of skulls and candles (29) Pool of blood 1 (79) Pool of blood 2 (80) Pool of blood and flesh (24) Brains (81) Skull on pole (27)	
		<b>Броня</b>		<b>Фонтаны частиц</b>
		Armour Bonus (2015) Green Armour (2018) Blue Armour (2019)		Черный (9032) Белый (9033) Синий (9029) Зеленый (9028) Фиолетовый (9031) Красный (9027) Желтый (9030) Spark (искры) (9026)
		<b>Роверы</b>		<b>ZDoom</b>
		Berserk (2023) Computer Map (2026) Invisibility (2024) Invulnerability (2022) Light Amp. Goggles (2045) MegaspHERE (83) Radiation Suit (2025) Soulsphere (2013)		Custom Sprite (9988) Decal (9200) Deep Water (9045) Hate Target (9076) Map Marker (9040) Map Spot (9001) Map Spot (gravity) (9013) Path Node (=Patrol Point) (9024) PolyObj Anchor (9300) PolyObj Spawn (Hurts to touch) (9303) PolyObj Start Spot (9301) PolyObj Start Spot (раздавливающий) (9302) Puller (5002) Pusher (5001) Secret Trigger (9046)
<b>Телепорты</b>	Arachnotron (9050) Archvile (9051) Baron of Hell (9052) Cacodemon (9053) Chaingunner (9054) Demon (9055) Zombieman (9061) Shotgun Guy (9060) Hell Knight (9056) Imp (9057) Mancubus (9058) Revenant (9059)	<b>Ключи</b>		
Teleport Destination (14) Teleport with Z Height Gravity (9043) Teleport with Z Height No Gravity (9044)		BlueCard (5) BlueSkull (40) RedCard (13) RedSkull (38) YellowCard (6) YellowSkull (39)		
				<b>Звуки</b>
<b>Порталы</b>				Sector Sound Sequence 0-9 (1400-1409) General Sector Sound Sequence (1411) Ambient sound 1-64 (14001-14064) Окружающий звук 0 или звуки 0-255 (14065)
Lower Sector (9078) Upper Sector (9077)				
<b>События секторов</b>	<b>Невидимые монстры</b>			
Actor enters sector (9998) Actor hits ceiling (9996) Actor hits fake floor (9989) Actor hits floor (9999) Actor leaves sector (9997) Eyes above fake ceiling (9982) Eyes above fake floor (9992) Eyes below fake ceiling (9983) Eyes below fake floor (9993) Player uses sector (9995) Player uses wall (9994) Silent sector (9082)				
	<b>NPC-думеры</b>			
	Marine Berserk (9102) Marine BFG (9111) Marine Chaingun (9107) Marine Chainsaw (9103) Marine Fist (9101) Marine Pistol (9104) Marine Plasma Rifle(9109) Marine Railgun (9110) Marine RLauncher (9108) Marine Shotgun (9105) Marine SSG (9106) Scripted Marine (9100)			
<b>Мосты</b>				
Bridge Custom (9990) Bridge Radius 8 (5065) Bridge Radius 16 (5064) Bridge Radius 32 (5061)				