ZScript Basics:

A Guide for Non-Programmers

(from a non-programmer)

Contents

[Introduction 1](#_Toc38117737)

[Why ZScript? 1](#_Toc38117738)

[Classes instead of actors 1](#_Toc38117739)

[Anonymous functions 1](#_Toc38117740)

[Variables and data types 1](#_Toc38117741)

[Data types 1](#_Toc38117742)

[Pointers and casting 1](#_Toc38117743)

[Basic pointers 1](#_Toc38117744)

[Using pointers in ZScript 1](#_Toc38117745)

[Casting and custom pointers 1](#_Toc38117746)

[Type casting 1](#_Toc38117747)

[Custom functions 1](#_Toc38117748)

[Non-void functions and return values 1](#_Toc38117749)

[Virtual functions 1](#_Toc38117750)

[ZScript Virtual Functions 1](#_Toc38117751)

[Common virtual functions 1](#_Toc38117752)

[Event Handlers 1](#_Toc38117753)

[Arrays 1](#_Toc38117754)

[Dynamic arrays 1](#_Toc38117755)

[Flow Control 1](#_Toc38117756)

[Operators 1](#_Toc38117757)

[Checks 1](#_Toc38117758)

[Setters 1](#_Toc38117759)

# Introduction

Everyone is talking about ZScript. *DECORATE is dead*, they say. *You should be using ZScript*, they say. *I’m not gonna help you with your awful hacky DECORATE code,* they say.

You look around, confused. You’ve been using DECORATE for years now. You check ZDoom Wiki, and it’s still almost all about DECORATE. ZScript is barely documented. They tell you to look into ZScript code in gzdoom.pk3, and when you do, it makes your head spin. DECORATE is plain and simple, and with ZScript you don’t understand what’s going on at all.

Confusion leads to frustration, frustration leads to resentment. All the cool kids are playing with ZScript, and you don’t even know how and where to start. You wonder if your Doom modding career is over.

Sounds familiar?

That’s where lots of DECORATE users (some of them—known and respected modders even) have been finding themselves ever since ZScript became a thing. The reason is simple: ZScript is basically a programming language, and DECORATE has barely anything to do with that concept. As a result, there are *plenty* of people who’ve been doing just fine with DECORATE, and just a *handful* of people with previous programming experience who find it much easier to use ZScript. And it’s not surprising, ZScript *is* easier to use (more on that later) but it’s not easy for a non-programmer to begin.

I decided to write this short entry guide to help alleviate exactly that: it covers some basic programming concepts in simple terms which will help you to *get started*. Starting, after all, is the hardest part.

# Why ZScript?

Before talking about variables, pointers and classes it’s a good idea to answer this simple question: why use ZScript at all? What if you’ve been doing relatively simple things and DECORATE + occasionally ACS have been working out fine for you? Should you still switch?

You see, the question should actually be reversed. ZScript is now the default in GZDoom; DECORATE is deprecated (meaning it’s supported but will never be developed or updated further). So, you should ask yourself this: **are there reasons why I would *not* use ZScript?** Let’s take a brief look.

**ZScript is harder**

That’s incorrect. Not that you’re wrong, but that the statement itself makes little sense; ZScript is a *tool*, it can’t be hard or easy. How complex your code will be depends entirely on you. When you get the hang of it, ZScript is ultimately much easier to work with than DECORATE simply because the resulting code is cleaner and easier to read (and thus easier to get help with, by the way).

In other words, ZScript and DECORATE have the same skill floor, but ZScript has a much higher skill ceiling. How hard it is depends on how you use it and what you do with it.

**I’m used to doing stuff in DECORATE**

And you don’t have to let that go! Well, there are *some* DECORATE habits that you’d better eventually say goodbye to, but in essence you have to understand this: **DECORATE is fully included into ZScript**. There are some very simple syntax differences (such as having to use semicolons at the end of most strings), but otherwise you can write code in ZScript the same way you would do it in DECORATE. It can literally look identical.

**But if I’m going to write the same code I would in DECORATE, what’s the point of switching?**

It’s very simple: when you start using ZScript, eventually there will come a point when you decide to use a more advanced feature or create a custom function (or maybe even somebody will create it for you), or some other ZScript-only method. And when that moment comes, if you’ve been writing your code in ZScript up until that point, you’ll just be able to plug that new shiny feature into it. Whereas, if you’ve been using DECORATE, you’ll have to first at least partially translate your code into ZScript, which is *easy* but also long and tedious, and definitely not something you’re gonna want to do when the moment comes.

**I’ve done too much stuff in DECORATE already**

Switching mid-project can be difficult. But don’t fret! You don’t actually have to do that immediately.

First, DECORATE and ZScript can be combined within one pk3 just fine. (Not within one file, though; DECORATE and ZScript have to be placed separately inside the eponymous files.) In fact, you can even write a custom function in ZScript and then use it within a DECORATE actor. The only thing you have to consider is that DECORATE is loaded last, which means that DECORATE actors can inherit from ZScript ones, but not the other way around. Still, this means you don’t have to convert everything right away.

Second, converting your own stuff may be one of the best ways to learn. First you can do simple conversion where you port the code directly, only changing the syntax, then you’ll start poking here and there, optimizing and adding stuff. In no time you’ll know the basics of ZScript.

**But I’m using Zandronum!**

…Oh. That. Well, that can’t be helped, unfortunately. Personally, for me sacrificing the ability to use ZScript in favor of Zandronum features is not an option, hence I went GZDoom-only, but your approach may be different. However, as I’m writing this, Zandronum hasn’t updated in like 3 years. Even its DECORATE is an ancient version that doesn’t support simple things like FloatBobStrength.

In short, by using DECORATE you’re limiting yourself without gaining anything.

Still, even if it’s obvious that using DECORATE is pointless, it’s a good idea to take a look at some advantages of ZScript.

**Real variables**

If you’ve used ACS you probably have an idea of how variables work; otherwise there’s a chapter on them further on. The point is, any mildly complex DECORATE code is filled with dummy items that you use to check for various stuff. In ZScript you don’t have to do that because you can store that information in real variables attached to classes.

**No duplicates**

Your DECORATE code is likely filled with duplicate actors that replace various things, or actors that are basically identical but use a different function somewhere. A simple example would be special effects, like light halos. In ZScript you can say goodbye to duplicates because you’ll have tools at your disposal that allow to make the actor change according to conditions instead of making multiple versions of that actor. Case in point: you can create a single light halo actor and change its color upon spawning.

You also won’t have to duplicate the same code into multiple places.

**Custom functions and properties**

You can create custom functions in ZScript. You can even give them any name and it doesn’t have to start with "A\_". And then you can use those functions throughout the mod (and even in DECORATE code if you have any).

**Changing properties and flags on the fly**

In DECORATE you’re limited to "setter" functions such as A\_SetScale or A\_SetRenderstyle when you want to change a property, and A\_ChangeFlag for flags; thus, you can’t change anything that there isn’t a setter for.

In ZScript you can. You can (will need to in certain situations) use those setters in ZScript as well, but a good portion of properties can be changed at any point. For example, you can change the ammo type used by a weapon, or an actor’s size (height and radius). You can the majority of flags on the fly as well, altering actors’ behavior.

**Interaction with map data**

ZScript can obtain and even change certain map-related data. While there are some reasonable limits to this, you can create pretty interesting stuff with that, such as a projectile that bounces off surfaces at correct angles, or explosives that "destroy" certain doors.

**Reduced performance impact**

With ZScript you’ll have tons of tools at your disposal to make your stuff more efficient. Even if you don’t do anything special, when you just follow some good basic rules of syntax and [flow control](#_Flow_Control) (more on those later), your code will *already* work much faster than something similar would in DECORATE.

# Classes instead of actors

Let’s talk basic syntax. The first thing you need to know is that DECORATE is used only to define actors—objects that "physically" spawn in the game world. These are monsters, decorations, weapons, inventory items, and player pawns (the actors you control in the game). There are some special cases, for example built-in GZDoom particles (the ones that spawn by default when you use A\_RailAttack or A\_CustomRailGun), which are not technically actors, but more on that later.

ZScript covers much more. ZScript also defines the HUD (fullscreen HUD, statusbar, etc.), menus (yes, even the main menu) and other stuff. The basic unit of ZScript is not an actor but a **class**.

A class is just a container for code (and that can be HUD code, menu code, actor code, etc.). An actor *is a type of class*. **Actor** is just a base class used by Doom actors; it contains the definitions for all properties, flags and functions available to Doom actors. Other common base classes are BaseStatusBar (contains all base code for HUD elements) and ListMenu (which, obviously, contains menu stuff).

As a result, in DECORATE a definition (i.e. a part of code) begins with the word Actor, because DECORATE can’t be used to code anything that isn’t an actor. In ZScript definitions begin with the word Class.

There are also some simple differences in syntax. Here’s a comparison of templates.

**DECORATE**:

#### Actor MyActor {

#### property1

#### property2

#### property3

#### +FLAGNAME

#### 

#### states {

#### Spawn:

#### SPRT A 1

#### loop

#### Death:

#### SPRT B 5 A\_Function

#### SPRT CD 5

#### SPRT E -1

#### stop

#### }

#### }

**ZScript**:

#### version "4.2.4" //you need to declare your ZScript version only once, at the top of your main zscript file

#### Class MyClass : Actor {

#### Default {

#### property1;

#### property2;

#### property3;

#### +FLAGNAME

#### }

#### states {

#### Spawn:

#### SPRT A 1;

#### loop;

#### Death:

#### SPRT B 5 A\_Function;

#### SPRT CD 5;

#### SPRT E -1;

#### stop;

#### }

#### }

*Note:* In case you don’t know, this is called pseudocode and it’s widely used as examples in programming manuals as well as by people. Pseudocode is a code that represents the way actual code would look but does not contain actual functions, properties, etc.

So, what are the differences here?

* Definitions begin with the word Class
* If you want to make a completely original actor, your class has to inherit from Actor (which is the base class for all actors); you can’t *not* inherit from anything
* Default properties and flags have to be enclosed in a Default { } block instead of just being written somewhere above states
* All lines except block names (Class, Default, states) and flags have to end with a semicolon. (Flags *can* end with a semicolon, but it’s optional.)
* Not shown in the example: flag prefixes are *not* optional in ZScript. I.e. for example, in +INVENTORY.AUTOACTIVATE you can’t omit INVENTORY.

Knowing just these points, you can already start coding in ZScript. Next, we delve into ZScript-only features.

# Anonymous functions

This isn’t really a ZScript-only feature because it was available in DECORATE before ZScript became widely available. However, that was a brief period and many DECORATE users missed that; plus, Zandronum’s version of DECORATE doesn’t support this feature at all.

The concept is very simple: an anonymous function is when you combine a bunch of functions inside {curly braces} and make it a code block. This way you can execute multiple functions but attach them to a single frame. So, for example instead of this:

TNT1 A 0 A\_GunFlash

TNT1 A 0 A\_Recoil(2)

TNT1 A 0 A\_SpawnItemEx("EmptyCasing")

TNT1 A 0 A\_FireBullets(5,1,1,0)

…You can do this:

TNT1 A 0 {

A\_GunFlash();

A\_Recoil(2);

A\_SpawnItemEx("EmptyCasing");

A\_FireBullets(5,1,1,0);

}

And that’s much cleaner and more convenient for multiple purposes.

There are a few things you need to remember when using anonymous functions:

* *Both* in DECORATE and ZScript all functions used inside anonymous functions **must** end with a semicolon, and, if you don’t specify any parameters, you still need to include empty parentheses at the end().
* You can use if/else and other types of conditions in anonymous functions.
* You can't use jump commands directly, and instead will have to use the return command

These aspects are described in more detail in the [**Flow Control**](#_Flow_Control) section.

# Variables and data types

If you’ve used ACS, you’re probably familiar with variables. Variables can be defined and used in ZScript in a similar manner, but there are more **types** available.

A variable is a piece of data that holds some information — a number, a string of text, coordinates, etc. When declaring variables, you give them names which you can use to reference them later. You can declare a variable in a class to use it anywhere in that class, or inside an anonymous function to use it in that function.

Here’s a simple example of declaring a variable in a class and using it:

Class SpeedyImp : DoomImp {

int speedups; // creates a variable that holds an integer value

Default {

health 300;

}

states {

Pain:

TNT1 A 0 {

if (speedups < 5) { // Check if the value of the variable is below 5

speedups++; // if true, increment by 1; it's the same as doing speedups = speedups+1;

speed \*= 1.2; // increase monster's speed by 1.2; same as doing speed = speed \* 1.2;

}

}

goto super::Pain; // go to the normal Pain state

}

}

*Note*: **speed** here is the **speed** property that all actors can use, not a custom variable.

Whenever this Imp is hurt, it'll increase it's speed by x1.2 but until up to 5 times.

The full syntax for declaring variables is as follows:

[access modifier] [transient] <type> <name> [scope]

In the vast majority of cases you won't need all of those elements. Most commonly your variables will look something like this:

string foo; //a variable that holds a string

protected int bar; //a variable that holds an integer number

* **access modifier** (optional)\* — protected, private, or static
  + protected — this variable can be changed only from this class and classes inheriting from it but it can’t be changed from anywhere else
  + private — this variable is only available to this class and nothing else
  + If keywords are unspecified, the variable can be changed from anywhere in the game provide you have a pointer to the class that contains it (see [Pointers and Casting](#_Pointers_and_casting))
* transient (optional) means the value won’t be stored in saved games
* **type** — the type of *data* that a variable can hold: an integer number, a float-point number, a string, etc. (See below for the list of common types)
* **variable name** — a case-insensitive line of text that you’ll use to access the variable and its value
* **scope** (optional) — ui, clearscope, virtualscope, or play — defines in which *context* this variable is accessible. See more in the **Scope** chapter. You won't need it in most cases.

\*In general, if you're creating a custom actor with custom variables and you want to make sure they can't be changed by anything except this actor, it's a good idea to make them private or, if it has child actors that need to use the same variables, protected.

## Data types

Common data types that variables can use:

* **int** — holds an integer number (such as 1, 2, 3, 10, 500, etc.)
  + Many existing properties are also integer values, for example damage and health. That’s why projectiles can’t deal 2.5 points of damage, only 2 or 3.
* **double** — holds a float-point number (such as 2.5). Note that float is also an existing type, but double is used in ZScript instead because it’s essentially the same thing but it has higher precision. That’s usually not something you need to worry about as a user, just remember that what’s called **float** in ACS is **double** in ZScript.
  + There’s a whole lot of values in GZDoom that are doubles. For example, an actor’s angle, pos.x, pos.y, pos.z, height, radius, speed, alpha, bouncefactor are all doubles.
* **bool** — a boolean variable holds a true/false value. You can set and check it against true and false, such as if (foo == true) (to check) and foo = false; (to set).
  + The most common example of a bool is actor flags. While flags in the Default block are a special case, since you can set them with + and -, those flags are internally connected to boolean variables named bFLAGNAME. You can change the majority of flags on the fly by using those names; for example, you can do bSHOOTABLE = false; to suddenly make an actor unshootable.
  + No quotation marks! "True" is a string holding the text "True", while true is a boolean value.
  + Note, **true** and **false** are actually just integers 1 and 0 respectively, but it’s highly recommended to actually use true and false when checking/setting a bool, so that you know this is a bool and not an int just by looking at it.
  + A shorthand for if (foo == true) is simply if (foo). And if (foo == false) can be replaced with if (!foo) (! means "not" and inverts any check).
* **string** — holds *case-sensitive* text (such as "Adam")
  + Setting and changing it requires using double quotes: string foo = "Bar"; creates a variable foo that holds the text "Bar".
* **name** — holds *case-insensitive* text (i.e. ‘adam’, ‘Adam’ and ‘ADAM’ are all the same)
  + In contrast to strings setting and changing them can be done by using *single* quotes. You can still use double quotes, but it’s a good idea not to do that, so that when you look at the variable, you’ll immediately know it’s a name and not a string. In fact, for custom variables in the majority of cases it’s better to use a name than a string, since there are relatively few applications for case-sensitive text.
  + A bunch of stuff in GZDoom are names, for example class names and values of various properties such as Renderstyle or Bouncetype.
* **vector2** — holds global (map-wise) 2D coordinates of something, i.e. an object’s pos.x and pos.y; or an object’s velocity in 2D space, i.e. the object’s vel.x and vel.y. The contents of this type of variable is two float-point values enclosed in parentheses, such as (15,14.2).
* **vector3** — similar to vector2, but for 3D space, so it can hold pos.x, pos.y and pos.z, or, in case of velocities, vel.x, vel.y and vel.z. An example of a vector3 expression that you can use it your code is pos which holds the coordinates of the actor it’s called from. For example you can do Spawn("DoomImp",pos); to spawn an Imp at exactly the location of the actor that calls this function. (Spawn is an internal ZScript spawning function.)
  + If you have only a vague understanding of how coordinates work and why they’re called "vectors", it’s very simple: every map has an origin point with coordinates of (0,0,0). Any object within a map has coordinates *relative* to that origin point; for example, if an actor is positioned at (15,12.3,0) that means it’s located 15 units to the east, 12.3 units to the north and 0 units vertically off the map origin point. Since all coordinates are *relative* to that (0,0,0) origin, this makes all coordinates *vectors*; a vector in this context is basically a line that starts at (0,0,0) and ends wherever the object is.
  + Similarly, velocity is just how quickly an object is changing position per tic (1/35 of a second). If an actor’s velocity is (15,0,1.2), every tic it moves 15 units north, 0 units east/west and 1.2 units upward. Basically, actor movement is their vector3 of velocity being constantly added to their vector3 position.
* **Class<Actor>** — a variable that holds a name of an actor.
  + The <Actor> part can be substituted for something else, if you want to limit this variable to being able to hold a pointer to only something specific, for example Class<Ammo>.
* **actor** — a variable that holds an instance of an actor (i.e. a pointer to it). It’s not a name of an actor class, but a *pointer* to a *specific* actor that exists in the level. Learn more in [Pointers and Casting](#_Pointers_and_casting).
* **state** — holds a reference to a state (such as Spawn, Ready, etc.)

Variables are mostly either inside anonymous functions, or somewhere in the class but outside of any other block. Let’s expand on our speedy imp example used above:

Class WeirdImp : DoomImp {

int speedups; //this variable is available anywhere in the class

Default {

health 500;

}

states {

Pain:

TNT1 A 0 {

if (speedups < 10) {

speedups++;

double foo = frandom(0.8,1.2); // create a temporary variable s that holds a random value between 0.8 and 1.2

speed \*= foo; // multiply speed by that value

scale /= foo; // divide scale by the same value

}

}

goto super::Pain;

}

}

*Note:* Multiplying **speed** but dividing **scale** will result in the monster's speed and scale being inversely related: the bigger it is, the slower it'll be, and vice versa.

This weird Imp will randomly change its speed and scale (inversely related) when being hit, up to 10 times. (Note, Pain state can be entered multiple times, for example, in case of a shotgun blast.)

The example above uses both a class-scope variable (speedups) and a temporary variable used inside an anonymous function (foo). The differences are very simple:

**Class-scope variables aka fields**

* Fields by default can be changed from anywhere (this class, inheriting classes, even other classes if they get access to it—see [Pointers and Casting](#_Pointers_and_casting)).
* Fields can’t be declared and receive a value at the same time; when you declare them, they receive a default value (0 for ints, 0.0 for doubles, empty string for strings and names), and then you have to modify it somewhere. Speedups in the example above is initially equal to 0 and it’s increased by 1 when the Imp enters its Pain state for the first time.
* Fields keep their value while the class exists. That’s why every time we do speedups++, it increases by 1 and will keep that value throughout whatever the imp does.
* Since fields can be accessed from multiple places, it’s a good idea to give them a sensible and understandable name.

**Local variables**

* Variables declared inside anonymous functions are available only within that function.
* They can be declared *and* given a value within the same line.
* Obviously, whenever the function is executed again, this variable will be re-declared and receive the value. That’s why double foo = frandom(0.8,1.2); will create a temporary variable foo equal to a random value between 0.8 and 1.2 every time Pain state is entered.
* Their names aren’t that important, since they won’t exist after the function stops executing. Usually something very short is used.

# Pointers and casting

## Basic pointers

One of the primary concepts you need to have a good grasp on to use ZScript is pointers. A **pointer** is, in essence, a type of variable that gives you *access* to something—usually an actor.

DECORATE actually has pointers! But you are limited to using three of them: **master, target** and **tracer**. You’re probably familiar with them, but here’s a quick recap:

* Target is the most common pointer and it’s automatically used by monsters and projectiles:
  + In case of **monsters** a target is literally their current target—the actor they’ll be chasing and attacking (if there is one). Monsters acquire a target by calling A\_Look, then chase it with A\_Chase, and they aim at the target with A\_FaceTarget.
  + In case of **projectiles** a target is (counter-intuitively) the **shooter** of the projectile. So, if it’s a player-spawned projectile, the player pawn will be its target. Why is it even tracked? Because the shooter has to get **kill credit**: it allows the game to track how many monsters the player killed, who killed whom in multiplayer, and print out obituary messages (such as "*Playername* stood in awe of Spider demon"). If for some reason the projectile loses its target pointer (which normally shouldn’t happen), the killer won’t get the credit. (There are other more obscure mechanics involved; for example, a projectile can’t hit its shooter as long as the shooter is the projectile’s target).
    - *Note*: if you’re wondering if a projectile has any pointer to its actual target, i.e. the monster that it'll hit, the answer is no. Projectiles don’t need pointers to actors they hit because they simply hit whatever they collide with. (They do get a pointer to what they hit briefly when the hit happens, but you can't access it in DECORATE; more on that later.)
* Tracer pointer is normally only used by seeker projectiles, such as **RevenantTracer**. Projectiles using seeking functions such as A\_Tracer or A\_SeekerMissile continuously face their tracer to change their direction towards it.
* Master pointer is not set by anything in vanilla Doom, but you might’ve set it yourself via A\_SpawnItemEx which allows setting pointers manually via flags (SXF\_SETMASTER in this case).

Pointers in DECORATE can be set manually mostly with A\_SpawnItemEx by using the function’s flags. Doing this, you get access to functions such as A\_KillMaster or A\_RemoveChildren and such, which allow killing/removing actors from another actor that has a pointer to them. A\_FaceTarget is also a common example of a function that interacts with a pointer, making the actor face its target (if it has one).

## Using pointers in ZScript

In ZScript pointers are much more flexible. The first difference is how you use them: you can use them as prefixes for calling functions and setting properties on a specific actor. For example, doing alpha = 0.5; will change the translucency of the actor that calls this code, but doing master.alpha = 0.5; will change the alpha of the actor’s master.

You can use the same syntax to call functions on a specific actor from another actor, like so:

Class GraciousImp : DoomImp {

states {

Death:

TNT1 A 0 {

if (target) // same as if (target != null) — this checks that target exists before doing anything

target.GiveInventory("Shell",20); // GiveInventory is an internal version of A\_GiveInventory with basically identical syntax

}

goto super::Death;

}

}

*Note*: You may have noticed there are no curly braces around the **target.GiveInventory** block. You can do that when there’s only **one line** after the condition. In this case you can omit the braces; if there are 2 or more lines, you can’t do that.

This gracious Imp gives the target some shells when it dies (hence, if you killed it, it'll be you).

Now let's make something more advanced. We'll use a tracer pointer that is normally not used by monsters. But first, to make it a bit more colorful, we'll create a TRNSLATE lump and add some translations:

**TRNSLATE:**

BabyAngry = "0:255=%[0.85,0.00,0.00]:[2.00,1.96,1.39]" //a desaturated color translation that tints the actor red

BabyCalm = "0:255=%[0.05,0.01,0.84]:[1.39,1.96,2.00]" //a similar translation but it tints the actor blue

**ZSCRIPT:**

//This is a smaller version of Cacodemon that has x2 health and is blue

Class CacoBaby : Cacodemon {

Default {

health 800;

radius 16;

height 30;

speed 12;

floatspeed 6;

scale 0.5;

Translation "BabyCalm";

}

}

Class CacoDaddy : Cacodemon {

states {

Spawn:

TNT1 A 0 NoDelay A\_SpawnItemEx("Cacobaby",64,flags:SXF\_ISTRACER); //SXF\_ISTRACER sets the spawned actor as tracer; not to be confused with SXF\_SETTRACER

HEAD A 10 A\_Look;

wait; //wait loops the last frame instead of the whole state, in contrast to loop

Death:

TNT1 A 0 {

if (tracer) { //don't forget to double-check that tracer still exists

tracer.A\_PlaySound("caco/active"); //play the classic Cacodemon activation sound on the tracer

tracer.A\_SetTranslation("BabyAngry"); //change translation to the one defined in TRNSLATE lump

tracer.speed \*= 2; //multiply tracer's speed by 2

tracer.floatspeed\*= 1.5; //multiply tracer's floatspeed by 1.5

tracer.bNOPAIN = true; //set tracer's NOPAIN flag to true: will never enter its Pain state

}

}

goto super::Death;

}

}

*Note*: Don't forget that you have to use **NoDelay** if you want to do something in the very first frame of the Spawn state. Otherwise Doom skips that function.

The daddy Caco spawns a baby Caco when it appears, and makes the baby its tracer. When the daddy dies, it checks if its tracer still exists, and if so, does a bunch of stuff **on the tracer**: plays a sound, changes its translation and speed, and removes its ability to feel pain. The baby is out for blood.

We use tracer. as a prefix to execute functions on it and change its properties. **Also note, it's very important to null-check all pointers you use**: if you don't do that and for some reason the pointer is null (the actor doesn't exist), the game will close with an error (which is called a VM abort). A simple example why it could happen here is that the daddy spawns its baby 64 units in front of itself; if the daddy Caco is initially placed facing some other actor or a wall, it won't spawn the baby at all.

## Casting and custom pointers

But castingand custom pointers is where the actual fun begins. **Casting** is creating a variable and attaching something to it (usually an instance of an actor). In other words, casing basically means creating a custom pointer. There are two main cases when you need to use casting:

* To create a custom pointer that doesn't take place of master, target or tracer. As I mentioned earlier, you should avoid using these pointers when you can, since there's a lot of implicit behavior attached to them (for example, monsters will target their attacks at their target pointer).
* To get access to **class-specific variables**, which includes your custom variables. This concerns any custom variables you may have created as well.

First, creating the pointers. Just like any variables, they can be class-wide or local. Let's modify our daddy Cacodemon slightly:

Class CacoDaddy : Cacodemon {

actor baby; //create a variable baby (notice its type is actor)

states {

Spawn:

TNT1 A 0 NoDelay {

baby = Spawn("CacoBaby",pos); //Spawn is an internal function: Spawn(classname,position) where position is a vector3 holding coordinates

}

HEAD A 10 A\_Look;

wait;

Death:

TNT1 A 0 {

if (baby) {

baby.A\_PlaySound("caco/active");

baby.A\_SetTranslation("BabyAngry");

baby.speed \*= 2;

baby.floatspeed\*= 1.5;

baby.bNOPAIN = true;

}

}

goto super::Death;

}

}

*Note*: **pos** is an expression that simply contains the actor's own coordinates. We use it as a second argument of **Spawn** to spawn CacoBaby at CacoDaddy's position.

The behavior barely changes, but we're now using a custom pointer baby instead of pre-existing tracer. This frees up the tracer pointer to be used somewhere else (perhaps by one of the existing functions, who knows).

What exactly happens: baby = Spawn("CacoBaby",pos) spawns an actor named CacoBaby at the position pos (CacoDaddy's position) *and* casts CacoBaby to the variable baby.

You may wonder why we're not using A\_SpawnItemEx here. That's because ZScript Spawn function not only spawns an actor but also gives us its name—as a result, we can immediately cast it to the variable. A\_SpawnItemEx, however, spawns an actor but does not return any data. (See [Custom Functions and Data Types](#_Custom_Functions_and) to learn more about return values.)

One minor downside is that Spawn uses global offseets, not relative, so we can't spawn CacoBaby 64 units in front of [. But that's not a problem, since we can spawn it and then immediately move it using Warp (a ZScript function similar to A\_Warp):

Spawn:

TNT1 A 0 NoDelay {

baby = Spawn("CacoBaby",pos);

if (baby) //don't forget to immediately null-check the pointer!

baby.Warp(self,64,0,0); //moves the spawned baby 64 units in front of self (CacoDaddy)

}

HEAD A 10 A\_Look;

wait;

*Note:* For this simple example, we're not checking the position here at all, so if CacoDaddy was in front of a wall, the baby can end up inside a wall.

Self, as you probably already guessed, is a pointer to the current actor; since we're calling this from CacoDaddy, self is CacoDaddy. The full syntax for Warp is **Warp(pointer, xoffsets, yoffsets, zoffsets)**, and the offsets are relative, just like with A\_Warp, so we move the spawned baby 64 units in front of self (CacoDaddy). (Self is an existing pointer, you don't need to define or cast it.)

Now, we can go even further with this. Instead of using two different actors, we can use only one and modify it on the fly to make it look different:

Class CacoSingleDad : Cacodemon { //don't try to use **replaces Cacodemon**, or they'll be spawning each other continuously

actor baby;

states {

Spawn:

TNT1 A 0 NoDelay {

baby = Spawn("Cacodemon",pos);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

HEAD A 10 A\_Look;

wait;

Death:

TNT1 A 0 {

if (baby) {

baby.A\_PlaySound("caco/active");

baby.A\_SetTranslation("BabyAngry");

baby.speed \*= 2;

baby.floatspeed\*= 1.5;

baby.bNOPAIN = true;

}

}

goto super::Death;

}

}

*Note:* Some properties, such as **speed** can be set directly on an actor, but others are read-only and require a "setter" function, such as **A\_SetSize**. If you try to modify something, but GZDoom tells you that "expression must be a modifiable value", this often means you can't modify that value directly, look for a setter function.

By doing the above, we spawn the baby Cacodemon and immediately set all of properties: health, speed, translation, etc. Obviously, now you can't use this CacoSingleDad to directly replace Cacodemons, because if you do that, you'll end up with an endless cycle of CacoSingleDads spawning each other. There are several ways we could go around it. For example, we could do this in Spawn:

Spawn:

TNT1 A 0 NoDelay {

**if (!master) {** //the following block will only execute if the actor does NOT have a **master**

baby = Spawn("Cacodemon",pos);

if (baby) {

**baby.master = self;** //we set the spawning actor as the master of the spawnee

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

}

HEAD A 10 A\_Look;

wait;

…But while this method *can* be useful in certain situations, in this case we can do it much simpler:

Spawn:

TNT1 A 0 NoDelay {

baby = Spawn("Cacodemon",pos,**NO\_REPLACE**);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

HEAD A 10 A\_Look;

wait;

NO\_REPLACE flag of the Spawn function spawns an actor, blocking attempts to replace it. This is what BossBrain in Icon of Sin uses to spawn Rocket explosions that always look the same even if a mod replaces the Rocket actor.

Actually, let's take a look at that!

private static void BrainishExplosion(vector3 pos) //defines a private function that **BossBrain** uses internally to create the explosion effect

{

Actor boom = Actor.Spawn("Rocket", pos, NO\_REPLACE); //Without going into details, here **Actor.Spawn** is essentially the same as just **Spawn**

if (boom)

{

boom.DeathSound = "misc/brainexplode"; //changes Rocket explosion sound to what Icon of Sin uses

boom.Vel.z = random[BrainScream](0, 255)/128.; //randomizes vertical velocity of the explosion

boom.SetStateLabel ("Brainexplode"); //sets **Rocket** to **Brainexplode** state. Yes, rockets have a state like that!

boom.bRocketTrail = false; //disables rocket trail which GZDoom spawns by default

boom.SetDamage(0); //disables collision detection which is not wanted here

boom.tics -= random[BrainScream](0, 7); //changes the length of the sprite frame randomly

if (boom.tics < 1) boom.tics = 1; //makes sure that the length of the frame doesn't end up below 1 tic, otherwise it'll be skipped

}

}

We haven't yet covered creating custom functions (like BrainishExplosion above) or [s (like [BrainScream]), but you should be able to mostly understand what's happening by now: the function creates a rocket, changes its explosion sound, disables rocket trail and damage and slightly randomizes its animation speed. On the whole, Icon of Sin's death effect is more complicated than that (and it only works at specific map coordinates, by the way), but you get the gist.

## Type casting

There's one other case of casting that you'll need to use when working with classes that use custom variables or functions.

Let's say we want to make a version of Baron of Hell that drops a big Soulsphere when it's killed: this Soulsphere should set our health to 300 instead giving 100 HP limited to 200. Of course, we could create a new Soulsphere actor, but since we now know about casting, we try do this:

Class PrinceOfHell : BaronOfHell {

states {

Death:

TNT1 A 0 {

actor orb = Spawn("Soulsphere",pos);

if (orb) {

orb.amount = 300;

orb.maxamount = 300;

orb.pickupmessage = "Overcharge!";

orb.scale = (1.5,1.5);

}

}

goto super::death;

}

}

But if you run the code above, you'll get "Unknown identifier" script errors about amount, maxamount and pickupmessage.

The reason is simple: we're casting Soulsphere as **actor**, but properties like amount and maxamount are *not* defined in the Actor class; they're actually defined in the Inventory class. To achieve that, we need to cast it explicitly as Inventory. And this is what's called **type casting**:

Class PrinceOfHell : BaronOfHell {

states {

Death:

TNT1 A 0 {

inventory orb = Inventory(Spawn("Soulsphere",pos));

if (orb) {

orb.amount = 300;

orb.maxamount = 300;

}

}

goto super::death;

}

}

In this case inventory orb creates a variable orb of type Inventory, then casts it to an Inventory class and spawns it. You'll need to use this method whenever you're trying to get access to variables, properties and functions defined only for a specific class.

You can simplify type casting by using the word let:

let orb = Inventory(Spawn("Soulsphere",pos));

Let automatically sets the variable's type to what you're casting it to: in example above orb will automatically be cast to Inventory. Usually there's no reason not to use that, since it's very convenient, but manually specifying the variable type arguably makes the code more readable: you'll be able to immediately tell what type it is.

You'll need to use type casting for your own custom actors and their functions as well. Let's take a more advanced example: say, you created a sprite light halo and you want to attach it to torches. You have 3 versions of a halo (red, green, blue) and you don't want to define separate actors for each; instead you want to have only one actor and you want it to change its color depending on which torch spawned it. You can do this:

Class UniversalLightHalo : Actor {

name halocolor; //this variable will hold the color

Default {

+NOINTERACTION //**NOINTERACTION** makes an actor less resource-intensive and disables collision and gravity. Use it whenever possible

Renderstyle 'add';

alpha 0.35;

scale 0.5;

}

states {

Spawn:

TNT1 A 0 NoDelay {

if (halocolor == 'red')

sprite = GetSpriteIndex("HRED"); //**sprite** lets you manually set which sprite to display; it needs GetSpriteIndex to find the sprite by its name

else if (halocolor == 'blue')

sprite = GetSpriteIndex("HBLU");

else if (halocolor == 'green')

sprite = GetSpriteIndex("HGRN");

}

#### A -1; //**####** means "use previous sprite" — in this case it'll be HRED, HBLU or HGRN depending on the check above

stop;

Load: // When sprites are set via GetSpriteIndex, they need to be defined somewhere in the code, so that they're loaded into memory;

HRED A 0; // this fake state (which must NEVER be entered, obviously) serves that function just fine.

HBLU A 0; // Note you don't actually need to define them here, they can be put inside any other random Actor in your mod.

HGRN A 0;

stop; //stop isn't really required here since this state will never be entered

}

}

Class CustomRedTorch : RedTorch replaces RedTorch {

states {

Spawn:

TNT1 A 0 NoDelay {

let halo = UniversalLightHalo(Spawn("UniversalLightHalo",(pos.x,pos.y,pos.z+48)));

if (halo)

halo.halocolor = 'Red';

}

goto super::spawn;

}

}

Class CustomGreenTorch : GreenTorch replaces GreenTorch {

states {

Spawn:

TNT1 A 0 NoDelay {

let halo = UniversalLightHalo(Spawn("UniversalLightHalo",(pos.x,pos.y,pos.z+48)));

if (halo)

halo.halocolor = 'Green';

}

goto super::spawn;

}

}

Class CustomBlueTorch : BlueTorch replaces BlueTorch {

states {

Spawn:

TNT1 A 0 NoDelay {

let halo = UniversalLightHalo(Spawn("UniversalLightHalo",(pos.x,pos.y,pos.z+48)));

if (halo)

halo.halocolor = 'Blue';

}

goto super::spawn;

}

}

*Note:* The names of the sprites, as well as **alpha** and **scale** values are, obviously, just examples here.

Using the code above each of the torches will spawn a halo and immediately set its halocolor variable to whatever you need.

# Custom functions

Sometimes you need to perform a bunch of similar actions in multiple places and/or multiple actors. Usually you can simplify things by creating a custom function using the following syntax:

[access modifier] [virtual] [action] <type> <name> ([arguments]) [scope]

These are mostly the same as with variables, with some notable differences:

* **access modifier** (optional)\* — protected, private, or static
  + protected — this function can be called only from this class and classes inheriting from it but it can’t be called from anywhere else
  + private — this function is only available to this class and nothing else
  + If keywords are unspecified, the variable can be changed from anywhere in the game provide you have a pointer to the class that contains it (see [Pointers and Casting](#_Pointers_and_casting))
* **virtual** (optional) means the function can be overridden so that you make it do different things (read on to learn about virtual functions)
* **action** is a completely optional keyword that is only needed for weapons. Weapon functions are peculiar, because you call them from weapon actors, but the function is actually executed on the *player pawn*: for example, A\_FireProjectile is normally called from a weapon, but it's the player's model that actually spawns the projectile. More
* **type** — data type; types are the same as variables, but you can also use void if the function just does stuff, but retrieves no data (see below)
* **name** — a case-insensitive name of the function
* **(arguments)** — you can supply arguments to use in the function (such as in A\_FireProjectile you can pass the type of projectile, spawnheight, etc.). Can be empty.
* **scope** (optional) — ui, clearscope, virtualscope, or play — defines in which *context* this variable is accessible. See more in the **Scope** chapter. You won't need it in most cases.

Let's revisit our **CacoSingleDad** for an example:

Class CacoSingleDad : Cacodemon replaces Cacodemon {

actor baby;

void SpawnBaby() { //defines a function that can be used by CacoSingleDad

baby = Spawn("Cacodemon",pos,NO\_REPLACE);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

void AngerBaby() {

if (baby) {

baby.A\_PlaySound("caco/active");

baby.A\_SetTranslation("BabyAngry");

baby.speed \*= 2;

baby.floatspeed\*= 1.5;

baby.bNOPAIN = true;

}

}

states {

Spawn:

TNT1 A 0 NoDelay SpawnBaby();

HEAD A 10 A\_Look;

wait;

Death:

TNT1 A 0 AngerBaby();

goto super::Death;

}

}

The code should be simple enough to understand: we created two functions, SpawnBaby and AngerBaby. The first one spawns the baby Caco and performs all the stuff we need to do (sets color, health, speed, etc.), and the second one checks if baby exists and does the other stuff.

There are two characteristics of this function to consider:

* This is a **void** function as specified by void before its name: this means the function has no **return value**. In other words, when we call this function, it simply *does stuff* but we can't use it to *obtain any data*. (Read on to learn more about return values.)
* This function has no **arguments**—i.e. nothing that we can add inside the parentheses.

As a result, this function isn't very flexible, and most cases you'll need to create something more complex.

For example, let's say we want to create a universal "baby-spawning function" that allows us to manually specify the baby's class and health. For that we need to declare [variables](#_Variables) that will serve as arguments and then use them within a function:

#### //defining the function:

void SpawnBabyFlexible(Class<Actor> **spawnclass**, int **babyhealth** = 100, int **babyspeed** = 10, double **babyscale** = 0.5) {

baby = Spawn(**spawnclass**,pos,NO\_REPLACE);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(**babyhealth**);

baby.speed = **babyspeed**;

baby.A\_SetScale(**babyscale**);

baby.A\_SetTranslation("BabyCalm");

}

}

#### //using the function:

TNT1 A 0 NoDelay SpawnBabyFlexible("Cacodemon",800,12);

Pretty simple: to create arguments, define variables within the parentheses, using syntax that should already appear familiar:

type name = value

If you provide a **value**, this will be the **default value** for the corresponding argument. For example, int babyspeed = 10 defines **babyspeed** as an optional argument: if you don't set a value when calling this function, then the default value will be used (which in the example above is 10).

The arguments you defined can be used within a function as shown in the example, e.g. baby.A\_SetHealth(babyhealth); will make the spawned actor's health equal to the number you provide when calling the function (in the example it's 800). We also provided a default value of 100 for the argument.

We could make this function a bit more universal by improving how it treats the default values:

void SpawnBabyUniversal(Class<Actor> spawnclass, int babyhealth = **0**, int babyspeed = **0**, double babyscale = **0**) {

baby = Spawn(spawnclass,pos,NO\_REPLACE);

if (baby) {

baby.Warp(self,64,0,0);

if (babyhealth **!= 0**) //executes if **babyhealth** is *not* equal to 0

baby.A\_SetHealth(babyhealth);

if (babyspeed **!= 0**)

baby.speed = babyspeed;

if (babyscale **!= 0**)

baby.A\_SetScale(babyscale);

baby.A\_SetTranslation("BabyCalm");

}

}

In this version of the function the default values are 0 and they're treated as an instruction to use default values for the actor: unless you specify values for health, speed and scale explicitly, the spawned actor will simply use its own default values. So, for example, if you spawn a Mancubus instead of a Cacodemon, it'll have Mancubus' default health value; same goes for scale and speed.

It's up to you which arguments to make optional and which to leave compulsory (in SpawnBabyUniversal above the only non-optional argument is spawnclass—the name of the actor to spawn); the only rule to remember is that non-optional arguments should be defined *first*—you can't have non-optional arguments defined after optional ones.

You can notice that this function doesn't actually cover everything we used in CacoSingleDad originally (such as modifying the baby's floatspeed), but you can easily expand it. It will also only work on actors that actually have a baby variable defined—so, if you want to create multiple custom functions, it may be a good idea to put them all inside your own version of the base Actor class and have your classes inherit from it.

It's also important to note that this function can't be static: it uses baby pointer that is only defined within CacoSingleDad and is not available to other classes.

A note on **function names**: you can give your functions any names you like, but in general it's a good idea to not use DECORATE's naming convention (A\_FunctionName) just avoid confusing yourself: this way you'll be able to tell at a glance that it's not a DECORATE function. However, it *can* be a good idea to use some sort of a custom prefix, such as your mod's initials.

## Non-void functions and return values

Void functions are the functions that do stuff and don't return any data. But there are many cases when you need to retrieve some sort of data using a function. Here's a very basic example:

int GetTargetHealth() {

if (target)

return target.health;

else

return null;

}

This function checks if the actor that called it has a target, and if so, returns the amount of health it has. If there's no target, it doesn't return anything. (null means no data; note that it's *not* the same as zero, it's literally nothing—if you try to print it out, you'll either get nothing or useless garbage data)

Note that you *have to* include a null-check here, as well as cover all possible cases: there has to be a return value for the case where target doesn't exist, that's why we must provide return null. If we don't, the function won't know what to do. (Nothing terrible will happen, GZDoom simply won't start. Bad stuff will happen if you skip the null-check, however, since it can cause a VM abort due to a null pointer).

It can also be slightly simplified:

int GetTargetHealth() {

if (target)

return target.health;

return null;

}

We don't need to use else here because the function cuts off at the point where you use return; so, if the target exists, the function will return its health and immediately stop doing anything else.

Function types are the same as variable types: they can hold numeric values, pointers, coordinates, etc. Here's an example of a custom version of A\_MonsterRefire that is a state function:

Class ChaingunGuyWithAMagazine : ChaingunGuy {

int monsterclip; //this custom variable holds the number of "ammo" in monster's magazine

override void PostBeginPlay() { //in PostBeginPlay we set the initial number of "ammo" to be 40

super.PostBeginPlay();

monsterclip = 40;

}

state CustomMonsterRefire(int ChanceToEnd = 0, statelabel endstate = "See") {

if (monsterclip == 0) //check how much "ammo" is left

return ResolveState("Reload"); //if it's 0, goto Reload state

else if (ChanceToEnd > random(0,100)) //otherwise check ChanceToEnd against a random value

return ResolveState(endstate); //if true, go to end state

return ResolveState(null); //otherwise don't do anything

}

states {

Missile:

CPAS F 2 A\_CPosAttack;

CPAS E 2 A\_FaceTarget;

CPAS F 2 A\_CPosAttack;

CPAS E 2 A\_FaceTarget();

TNT1 A 0 A\_MonsterRefire(5,"AttackEnd");

loop;

AttackEnd:

CPAS A 20;

goto See;

Reload:

CPAS A 40 {

monsterclip = 40;

}

goto See;

}

}

This function is designed for monsters that have "magazines" and need to reload their weapons. The monster in question doesn't use actual items for ammo, instead there's a variable that holds how much ammo it has.

This function works as a state jump, such as A\_Jump: when the function is called in a monster's state, it tells the state machine where to go. Specifically, it does the following:

* Checks if the monster has run out of "ammo" (monsterclip is 0). If so, it returns "Reload" state where other stuff happens (monsterclip gets reset to whatever value is considered as full magazine).
* Otherwise it checks its argument ChanceToEnd (by default it's 0): if it's greater than 0, there's a randomized chance that the monster stops firing. In the example above the monster uses the value of 5.
  + It jumps to a state provided in the second argument of the function. By default, it's "See" but in the example above it's "AttackEnd". (I.e. this monster has a custom animation for stopping the attack, but other monsters using this function may not and they just jump to "See".)
* Finally, if none of the checks go through, the function returns nothing. ResolveState() is the correct way to return states in ZScript (you can't directly put in a state name). ResolveState will be covered in more detail in [Flow Control](#_Flow_Control_1).
  + In this case, if the function returns null, the state machine will continue going through the state. In the example above it'll show frame CPAS E for 1 tic and then it'll hit loop and go back to the beginning of the state.

## 

## Virtual functions

Virtual is a keyword that makes a function overridable. There are two primary uses for it.

First, you can apply it to your own functions, like so:

Class CacoSingleDad : Cacodemon {

actor baby;

virtual void SpawnBaby() {

baby = Spawn("Cacodemon",pos,NO\_REPLACE);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

}

Class SomeOtherCaco : CacoSingleDad {

override void SpawnBaby() {

actor a = Spawn("Cacodemon",pos,NO\_REPLACE);

if (a)

a.master = self;

}

}

**SomeOtherCaco** above completely redefines SpawnBaby function to do something else: it also spawns a Cacodemon but it doesn't attach it to baby pointer; instead assigns SomeOtherCaco as the spawned Caco's master.

However, this doesn't seem especially useful. What is done more commonly with virtual functions is that they're overridden to *add* some stuff:

Class CacoSingleDad : Cacodemon {

actor baby;

virtual void SpawnBaby() {

baby = Spawn("Cacodemon",pos,NO\_REPLACE);

if (baby) {

baby.Warp(self,64,0,0);

baby.A\_SetHealth(800);

baby.A\_SetSize(16,30);

baby.speed = 12;

baby.floatspeed = 6;

baby.A\_SetScale(0.5);

baby.A\_SetTranslation("BabyCalm");

}

}

}

Class SomeOtherCaco : CacoSingleDad {

override void SpawnBaby() {

**super.SpawnBaby();** //calls the original **SpawnBaby()** first

if (baby) {

baby.A\_SetScale(0.4);

baby.master = self;

}

}

}

Super is a pointer to the original function. In the example above SomeOtherCaco *first* does everything the original SpawnBaby() function does, and after that it adds some changes: it modifies the spawned baby's scale and sets itself as the baby's master.

As for the second point, it's important enough to be its own chapter:

# ZScript Virtual Functions

While **virtual** is just a type of function, the one that you can even use yourself ([as described above](#_Virtual_functions)), much more often you'll be using (overriding) the existing virtual functions.

The base **Actor** class has a lot of virtual functions attached to it which it calls under certain conditions *outside* of states. Overriding them allows to add a bunch of effects to your actors that don't have to (or can't) be bound to a specific state.

One of the most common virtuals you'll be using this way is Tick(): a virtual function that is called by all actors every game tick. It performs everything actors need to do continuously: changes positions, velocity, checks for collision and a bunch of other things. You can add your own effects into that function:

Class TemporaryZombieman : Zombieman {

Default {

renderstyle 'Translucent';

}

override void Tick() {

super.Tick(); //don't forget to call this! otherwise your actor will be frozen and won't interact with the world

A\_FadeOut(0.01);

}

}

This Zombieman will continuously (and relatively quickly) fade out as it exists.

Remember that Tick() is called even while the game is paused, so you should add a check for that:

Class TemporaryZombieman : Zombieman {

Default {

renderstyle 'Translucent';

}

override void Tick() {

super.Tick();

if (!isFrozen()) //using **!** before the check is the same as using **== false** after it. You can use **if(isFrozen() == false)** instead

A\_FadeOut(0.01);

}

}

IsFrozen() is a bool that returns true if the actor that calls it is currently frozen, which can happen when:

* the game is paused (by opening main menu, console or pressing Pause in single player);
* "freeze" cheat is entered in the console;
* the player has a PowerTimeFreezer powerup and the actor in question does *not* have a NOTIMEFREEZE flag.

There's a ton of things you can do this way. A common example when using Tick() is convenient is when your actor needs to continuously spawn some sort special effect every tick (such as a trail or an after-image). Here's a handy example of doing an after-image this way:

Class BlurryCacoBall : CacoDemonBall {

override void Tick() {

super.Tick();

if (isFrozen()) //first we check if the actor is frozen

return; //if so, we stop here and don't do anything else; everything below will only happen otherwise

actor img = Spawn("CacoBall\_AfterImage",pos); //you're already familiar with casting

if (img) {

img.A\_SetRenderstyle(alpha,GetRenderstyle()); //transfer the current actor's **renderstyle** and **alpha** to the spawned actor

img.sprite = sprite; //transfer current sprite to the spawned actor

img.frame = frame; //transfer current frame to the spawned actor

}

}

}

Class CacoBall\_AfterImage : Actor {

Default {

+NOINTERACTION //we don't need this after-image to be interactive in any way

}

states {

Spawn:

#### # 1 { //#### # allows us to use the same frame (as set by BlurryCacoBall earlier)

A\_FadeOut(0.05);

scale \*= 0.95;

}

loop;

}

}

The result is a pretty cool trail that is very easy to implement: notice, we didn't have to edit the states of BlurryCacoBall *at all*.

This principle applies to most virtual functions. Here's another example with PostBeginPlay(), a function that is called as soon as the actor is spawned and placed in the world but before its Spawn state starts:

Class MyActor : Actor {

int myvalue;

override void PostBeginPlay() {

super.PostBeginPlay();

myvalue = 10;

}

}

As explained earlier, when you declare class-scope variables, like myvalue above, you can't immediately give them a value. You either have to turn it into a property, or set that value somewhere. PostBeginPlay() is a good place to do that. Notice, that PostBeginPlay() is not like Tick(): it's called only once, so there's no need to check if the actor is frozen. If your actor has some sort of an attached "companion" actor (for example, a fireball that spawns an actor-based light flare around itself), it's also a good place to spawn them.

There are many, many other virtual functions that you will need to override. And remember: you won't always need to call **super** on them; sometimes you'll need to completely fill in what the function does, without calling its original version. Let's take a quick look at ModifyDamage() — an **Inventory** function used by protective items such as PowerProtection (a power-up that reduces incoming damage). This function gets the damage that is supposed to be dealt to the owner of the item, and then uses newdamage argument to tell the game how much damage to actually deal:

Class CustomProtection : Inventory {

Default {

inventory.maxamount 1;

}

override void ModifyDamage (int damage, Name damageType, out int newdamage, bool passive, Actor inflictor = null, Actor source = null, int flags = 0) {

if (inflictor.bMISSILE) {

newdamage = damage \* 0.5;

}

else if (inflictor.bISMONSTER && !inflictor.bFRIENDLY && inflictor.health > 0) {

newdamage = damage \* 0.1;

}

}

}

The overridden ModifyDamage() above first checks the source of the damage: whether it a missile or a monster itself (i.e. a monster's melee attack). For missiles the damage will be cut in half, while for monsters it'll be reduced by 90%.

ModifyDamage() gets a bunch of pointers, and we use them to decide what to do. Inflictor is an actor pointer to the object that dealt the damage directly (projectile, puff or a monster in case of a melee attack).

Notice that both Tick() and PostBeginPlay() are **void** functions (they have no return value) and they have no arguments. ModifyDamage() has arguments but it's also a void function. But that's not true for all virtual functions.

A good example of that is SpecialMissileHit() — an integer function that is called by projectiles when they collide with an actor. When a projectile collides with an actor, it calls SpecialMissileHit(), which returns an integer number that tells the projectile what to do:

* -1 (default) will make the projectile do what it does normally (explode, rip through if it has +RIPPER flag, etc.);
* 1 will make the projectile pass through the actor (it doesn't need a +RIPPER flag for that, it'll simply fly through instead of colliding);
* 0 will destroy the projectile (remove it completely without doing anything else).

Here's how it's used by MageStaffFX2 in Hexen:

override int SpecialMissileHit (Actor victim)

{

if (victim != target && !victim.player && !victim.bBoss)

{

victim.DamageMobj (self, target, 10, 'Fire');

return 1; // Keep going

}

return -1;

}

Notice that SpecialMissileHit() also gets a pointer victim of type actor: this is a pointer to the actor that was hit by the projectile. In the example above the projectile checks that the victim isn't the shooter of the projectile\*, or a player (any player) or a boss (has +BOSS flag); if so, it deals damage to the victim by calling DamageMobj() function (see below) and keeps going. If the victim *is* the shooter, or a player, or a boss, the projectile explodes.

**\*Note**: if you wonder why it's important to check that the victim isn't the shooter, it's because projectiles come out of the actor that shot them and *can* collide with it (and SpecialMissileHit() will be called even though normally projectiles can't hit their shooter).

As you can see, virtual functions are already attached to actors, and you mix your own stuff into them to add various effects. However, you can also *call* them just like you call regular actor functions. A common example of a function that you may often need to both override and call is DamageMobj():

**int DamageMobj (Actor inflictor, Actor source, int damage, Name mod, int flags = 0, double angle = 0)**

Called by the actor whenever it takes damage.

* **inflictor** - The actor pointer dealing the damage. Missiles are used here, with their owners being the *source*.
* **source** - The actor pointer which claims responsibility for the damage, responsible for causing infighting.
* **damage** - The amount of damage to deal out.
* **mod** - The 'means of death', or the damagetype.

(See the rest [on ZDoom wiki](https://zdoom.org/wiki/ZScript_virtual_functions#Actor))

The function is called on actors when they would receive damage (but before it's actually dealt). It gets a bunch of information, including the pointers to actors that deal the damage, and the raw damage value before it's modified by various resistances (as damage).

When the base DamageMobj() is called, it'll *deal* the damage. (As with other virtual functions, if you override it, then to call the *base* function you need to call super.DamageMobj).

Apart from dealing damage, it also *returns* an integer number: normally it should be the same as the amount of damage dealt.

Here's an example of how this override is used:

Class ZombieTroopman : Zombieman {

override int DamageMobj(Actor inflictor, Actor source, int damage, Name mod, int flags, double angle) {

if (source && source is "Zombieman")

return 0;

return super.DamageMobj(inflictor, source, Damage, mod, flags, angle);

}

}

This version of Zombieman checks whether the source of the attack was another Zombieman (or an actor inheriting from Zombieman). If so, it *doesn't* call super.DamageMobj and returns 0. In all other cases it deals damage normally and returns the amount of damage that was dealt.

As mentioned above, you can also *call* DamageMobj to—you guessed it—damage an actor. You can even do it from a DamageMobj override:

Class RetaliatingZombieman : Zombieman {

override int DamageMobj(Actor inflictor, Actor source, int damage, Name mod, int flags, double angle) {

if (source)

source.DamageMobj(self,self,damage,'normal'); //deals damage to the source—i.e. the enemy or player that damaged them

return super.DamageMobj(inflictor, source, Damage, mod, flags, angle);

}

}

This annoying Zombieman calls DamageMobj on the actor that dealt damage to them, and deals exactly the same amount of damage. Notice that, since there are no projectiles involved, both inflictor and source in this call are self, i.e. the Zombieman itself.

DamageMobj can be called from anywhere; in fact, it's actually probably the most common basic function used to deal damage.

Let's say you want to create a projectile that can pierce enemies and damage them, but don't want to use +RIPPER flag, since with this flag projectile will damage the enemy continuously, as it's flying through them. Instead, you want the projectile to always damage the enemy once and only once. That can be achieved with SpecialMissileHit we just talked about, and DamageMobj:

Class PenetratingBullet : FastProjectile {

actor hitvictim; //we'll use this custom pointer to store the last actor hit by the projectile

Default {

speed 85;

damage 0; //if this isn't 0, the projectile will deal damage immediately as it enters Death state

radius 2;

height 2;

scale 0.2;

obituary "%o was shot down.";

}

override int SpecialMissileHit(actor victim) {

if (victim && target && victim != target && victim != hitvictim) { //below is only performed if victim (the actor hit) is NOT the same as hitvictim (last actor hit)

victim.DamageMobj(self,target,10,'normal'); //deal exactly 10 damage to the victim

hitvictim = victim; //store the vicitm we just damaged as hitvictim

}

return 1; //keep flying

}

states { //we're just reusing Rocket sprites

Spawn:

MISL A 1;

loop;

Death:

TNT1 A 1;

stop;

}

}

Thanks to SpecialMissileHit we don't even need RIPPER. Instead of 10 you can, of course, supply any random expression you like as damage, for example 5\*random(1,8) will make it behave similarly to Doom, where projectiles deal randomized damage multiplied between 1 and 8.

Notice, that the inflictor in this case is self (the projectile itself), while the source is target— that is the projectile's target, which, as we remember is whoever shot the projectile.

### Common virtual functions

A non-comprehensive of some of the most common virtual functions you'll be overriding in your mods:

**Actor:**

* void Tick() — Called by all actors every tic to handle collision, movement and everything else.
* void BeginPlay() — Called after the actor is created, before any default properties are established. Can be used to set default values to custom variables. Do NOT destroy actors here!
* void PostBeginPlay() — Called after the actor is been created but before the first tic is played or any state called. A good place to do stuff like spawning another accompanying actors nearby (e.g. a lamp and a light halo), and anything else you'd normally do in the first frame of Spawn.
* bool CanCollideWith (Actor other, bool passive) — Called when two actors collide, depending on who ran into whom.
* int SpecialMissileHit (Actor victim) — Called by projectiles whenever they collide with an actor (including the shooter of the projectile!).

**Inventory:**

* void DoEffect() — Called every tic by inventory items that are inside an actor's inventory. Use it instead of Tick() to continuously do stuff on items.
* void AttachToOwner(Actor user) — Called by items when they are placed in an actor's inventory. After this call the user (the actor the item gets attached to) becomes owner, and the item can use the owner pointer.
* void DetachFromOwner() — Called anytime the item is fully removed from owner's inventory, whether by being tossed, destroyed or taken away entirely.
* void ModifyDamage (int damage, Name damageType, out int newdamage, bool passive, Actor inflictor = null, Actor source = null, int flags = 0) — Called by items capable of modifying the owner's incoming damage, such as PowerProtection.

A more detailed list can be found on the [ZDoom Wiki](https://zdoom.org/wiki/ZScript_virtual_functions#Actor).

# Event Handlers

We mentioned at the beginning of this guide that ZScript isn't restricted to actors and has other types of classes. One of the commonly used (and extremely handy) non-Actor classes is EventHandler. An event handler calls various virtual functions when certain events happen in the game and can be used as a replacement for some of the ACS scripts and much more.

To create an event handler, you need to define a class that inherits from EventHandler, and also add that class in MAPINFO, so the basic definition of any event handler looks like this:

#### //ZScript:

Class MyCustomStuffHandler : EventHandler {

#### //custom stuff goes here

}

#### //MAPINFO:

Gameinfo {

AddEventHandlers = "MyCustomStuffHandler"

}

By overriding virtual functions of an event handler, you can make stuff happen in the game without replacing actors, which allows to create universal mods with high compatibility, or produce effects that, for example, affect all monsters without the need to replace them.

Here's a simple event handler:

Class CorpseDestroyer : EventHandler {

override void WorldThingDied (Worldevent e) {

if (e.thing && e.thing.bISMONSTER);

e.thing.destroy();

}

}

Let's break it down how this works:

* All events of an event handler have access to a pointer e (the type of the pointer is WorldEvent). This pointer is a bit different from actor pointers we covered earlier; it's not a pointer to an in-game object, but rather to the *event itself*.
* Through pointer e you can access various other pointers that this specific event can access.
* Whenever anything in the world is killed, it triggers a WorldThingDied event. This event has access to the actor that was killed via e.thing pointer.
* In the example above we first check if e.thing exists (a standard null-check), and then we check if it has an ISMONSTER flag (which is normally the best defining feature of a monster).
* If both checks pass, we call destroy() on the thing to make it disappear from the map.

Notice that event virtual functions don't need a super. call as opposed to Actor virtual functions. (Other things that need to happen when something dies will happen anyway, it's not tied to our event handler)

So, this handler will remove anything that we kill. However, it's not very elegant, since every monster will just pop out of existence as soon as it dies—and it won't even finish its Death animation, it'll disappear as soon as its health reaches 0.

Let's say we want to fade it out. But we can't make it via an event handler—this event just called only once when the monster is killed, so we can't loop A\_FadeOut in it. For something like this Inventory objects are usually used as containers for special effects, like so:

Class CorpseFadeHandler : EventHandler {

override void WorldThingDied (WorldEvent e) {

if (e.thing && e.thing.bISMONSTER) //we check the thing that was killed exists and it's a monster

e.thing.GiveInventory("CorpseFader",1); //if so, give it this inventory token

}

}

Class CorpseFader : Inventory {

Default {

inventory.maxamount 1;

}

override void AttachToOwner (Actor user) {

super.AttachToOwner(user);

if (owner)

owner.A\_SetRenderstyle(alpha,Style\_Translucent); //after the item is attached, set owner's renderstyle to Translucent

}

override void DoEffect() {

super.DoEffect();

if (!owner)

return;

owner.A\_FadeOut(0.01); //phase the owner out

}

}

This is an easy and handy method to attach code to an actor without actually replacing the actor, which is something you might want to do if you're making a minimod that is meant to be universally compatible with other mods.

Let's take a look at a few other examples.

This handler could be used as a basis for a reward/score system:

Class RewardStuff : EventHandler {

int killedmonsters; //this will serve as a counter

override void WorldThingDied (worldevent e) {

if (e.thing && e.thing.bISMONSTER && e.thing.target && e.thing.target.player) { //check that the thing is a monster, and it was killed by the player

killedmonsters++; //increase the counter by 1

Console.Printf("Monsters killed: %d",killedmonsters); //prints out the number of monsters killed

if (killedmonsters >= 50) { //if it's 50 or greater...

Actor.Spawn("Megasphere",e.thing.target.pos); //spawn a megasphere under the player

Console.Printf("Here's a megasphere");

killedmonsters = 0; //reset counter

}

}

}

}

Handlers can be used to store global data, similarly to global variables in ACS. To retrive that data from a class, you'll need to cast your event handler just like you cast custom actors:

#### //don't forget to add this event handler via MAPINFO!

Class CheckMonsterAmount : EventHandler {

int alivemonsters; //a simple int variable that will hold the number of alive monsters

override void WorldThingSpawned (worldevent e) { //calls when an actor is spawned in the map

if (e.thing && e.thing.bISMONSTER && !e.thing.bFRIENDLY) //check if actor exists, is a monster and isn't friendly

alivemonsters++; //if so, increase variable by 1

}

override void WorldThingDied (worldevent e) { //calls when an actor dies in a map

if (e.thing && e.thing.bISMONSTER && !!e.thing.bFRIENDLY)

alivemonsters--; //decrease by 1

}

}

Class CyberdemonLeader : Cyberdemon replaces Cyberdemon {

override void PostBeginPlay() {

super.PostBeginPlay();

let event = CheckMonsterAmount(EventHandler.Find("CheckMonsterAmount")); //cast the event handler just like you cast actors

if (event) //check if cast was successful

A\_SetHealth(3000 + 100\*event.alivemonsters); //if so, change health value

console.Printf("cyberdemon health: %d",health); //a debug line: it'll print out how much health cybie has now

}

}

When spawned, this Cyberdemon will check the alivemonsters variable held in our custom event handler, then its health will be set to 3000 plus 100 health per each monster alive.

Finally, here's a slightly more advanced example where an event handler and a dummy item container are used to create a bleeding system:

Class BleedingHandler : EventHandler {

override void WorldThingSpawned (WorldEvent e) {

if (e.thing.player && !e.thing.FindInventory("PlayerBleedControl")) //check if the spawned thing is a player and it doesn't have the item in possession

e.thing.GiveInventory("PlayerBleedControl",1); //if so, give them the item.

}

override void WorldThingDamaged (WorldEvent e) { //When something is damaged...

if (e.DamageType != "bleed") //check if damagetype of the attack was "Bleed", otherwise do nothing

return;

if (!e.thing.FindInventory("PlayerBleedControl")) //if for some reason they don't have the item, also do nothing

return;

let bleeder = PlayerBleedControl(e.thing.FindInventory("PlayerBleedControl")); //Get a pointer to the PlayerBleedControl in the thing's inventory

if (!bleeder) //null-check the pointer

return;

bleeder.bleedbuildup = Clamp(bleeder.bleedbuildup + e.Damage, 0, 100); //If all above is true, increase 'bleedbuildup' by 'damage' (within 0-100 range)

if (random(1,100) < bleeder.bleedbuildup) { //Run a random 0-100 value against 'bleedbuildup'. If 'bleedbuildup' turns out to be higher...

bleeder.isbleeding = true; //make the thing bleed

bleeder.bleedsource = e.DamageSource; //store the source of damage (so that the killer gets credited if the thing bleeds out)

}

}

}

// The item that handles the bleeding itself: it holds the "gauge" and deals damage to the carrier when needed

Class PlayerBleedControl : Inventory {

Default {

+INVENTORY.UNDROPPABLE

+INVENTORY.UNTOSSABLE

inventory.maxamount 1;

}

bool isbleeding; // if this is true, you're bleeding

int bleedbuildup; // the "gauge"; we keep it between 0 and 100

actor bleedsource; // this stores who dealt the damage to the player

override void DoEffect () {

super.DoEffect();

if (!owner)

return;

Console.Printf("Bleed buildup: %d; Bleeding: %d",bleedbuildup,isbleeding); //prints out the buildup value (good for debugging)

if (level.time % 35 == 0) { // this will be true every 35 tics, i.e. once a second

bleedbuildup = Clamp(bleedbuildup - 1, 0, 100); // First of all, we reduce bleedbuildup by 1, still making sure it stays between 0-100

if (isbleeding) { // if we're already bleeding...

// We do the damage. The damage value is 20% of bleedbuildup value but always between 1-5. So, the fuller the gauge, the more damage you'll receive while bleeding.

owner.DamageMobj(owner,bleedsource,Clamp(bleedbuildup \* 0.2,1,5),"normal",DMG\_NO\_ARMOR|DMG\_THRUSTLESS|DMG\_NO\_ENHANCE);

}

if (random(1,80) > bleedbuildup) { //If a random value between 1-80 is higher than bleedbuildup we may stop bleeding

isbleeding = false; //The longer we bleed, the higher is chance we'll stop bleeding. If bleedbuildup is higher than 80, you can't stop bleeding.

}

}

}

}

*Note*: This is not a universal effect; puffs or projectiles need to have damagetype 'Bleed' in their properties to activate bleeding (since in most systems not all damage can cause bleeding). This, of course, can be easily edited: if if (e.DamageType != "bleed") check is removed, this will affect all cases when anything deals damage.

There are a few new things in this code worth pointing out:

* Clamp(value, min, max) allows modifying a value while making sure it doesn't exceed the min or max values. In the example above bleedbuildup = Clamp(bleedbuildup - 1, 0, 100) is the same as bleedbuildup -= 1, but it makes sure it never goes below 0.
* level.time is a global variable that returns how much time (in tics) has passed since the current map was started. It's a neat and simple way to make sure effects occur only after a specific period of time or with specific intervals (as above). It's necessary in constantly executing functions, such as Tick() or DoEffect(), since they don't have any analog of wait or delay.
* % is a [modulo operator](https://en.wikipedia.org/wiki/Modulo_operation): value1 % value2 will return the remaining number after a division of value1 by value2, known as modulus. For example, the expression 5 % 2 would evaluate to 1 because 5 divided by 2 has a quotient of 2 and a remainder of 1, while 9 % 3 would evaluate to 0 because the division of 9 by 3 has a quotient of 3 and leaves a remainder of 0; there is nothing to subtract from 9 after multiplying 3 times 3. Hence, the check if (level.time % 35 == 0) will return true every 35 tics.

# Arrays

## Dynamic arrays

You can think of arrays as an "expanded" version of variables: instead of holding one value, they can hold many of them in a list. Their definition begins with the keyword array, and otherwise is very similar to variables:

[access modifier] array < <type> > <name>[<size>] [scope]

*Note*: <angled brackets> are not part of the template, they're actually needed for the type, as shown in the example below.

A simple array would look like this:

array <Actor> actorlist[];

This creates an empty array that can hold pointers to actors, which you can later access by accessing this array. Let's take a brief look over some basics of using the arrays:

**arrayname.Push(pointer)** adds something (in our case — an actor) to an existing array:

let foo = Spawn("Zombieman",pos); //spawn a zombieman and cast a pointer to it

if (foo) //null-check the pointer

actorlist.Push(foo); //if pointer exists, push that actor into our array

**arrayname[index]** gives us a pointer to that entry in the array, allowing us to do something. [index] is a number, where 0 is the first entry in the array:

actorlist[0].destroy(); //destroys the first actor in the array

*Note*: don't forget to always check the pointer, in case that actor doesn't exist anymore.

**arrayname.delete(pointer)** deletes an entry from the array. Notice that it doesn't do anything to the object itself, it just deletes a *pointer* to the object:

actorlist.delete(actorlist[0]); //remove the first entry in the array from it. Array size will decrease by 1

# Flow Control

## Operators

Let’s take a quick look at the operators that let you change the values of variables, since you need to know how to actually use them.

### Checks

These operators allow you to check the value of variable and then do something, such as if/else blocks. Remember that conditions must not end with a semicolon! They start a new block that executes if the condition is met:

* == — "exactly equal to". Checks if the variable’s value is exactly equal to what you specify. Can be applied to int, double, string:
* if (deathsnum == 5)
* if (mystringvar == "Adam")
* if (vel == (0,0,0))
* < — "less than"
* > — "more than"
* >= — "more or equal to"
* <= — "less or equal to"
* ~== — "close to". For string variables it's a case-insensitive check. For doubles it makes the check slightly less precise (which means it'll use less processing power).

### Setters

* varname = value — make it equal to the value
* varname += value — make it equal to its current value + value. A longer version would be varname = varname + value
* varname++ — increase varname value by 1 if it’s an int. Same as varname = varname + 1
* varname-- — decrease varname value by 1 if it’s an int. Same as varname = varname – 1
* varname \*= value — multiply current value by value. Same as varname = varname \* value
* varname /= value — divide current value by value. Same as varname = varname / value