Decorate

Definition

DECORATE is a text-format lump which allows one to define actors that can be placed in a level. DECORATE was originally intended to aid in the creation of decorative objects such as the lamps and torches in Doom and other games without using up extra frames in DeHacked. However, it has been expanded to create virtually anything you want, not just decorations.

For historical reasons there are two distinct ways to define actors here. However, the old method has been completely superseded by the new one and it is strongly advised not to use it anymore.

DECORATE lumps are cumulative, meaning several may be loaded at once without overwriting each other in memory. Therefore, adding new actors does not require editing the pre-existing actor definitions. Instead, authors should simply create a new text-format lump (WAD format) or file (PK3 format) called DECORATE and define only their new actors there.

A full DECORATE reference can be found on the ZDOOM wiki.

Note: Not all of the DECORATE features are actually supported by Vavoom engine, some of them will be added in future releases.

Additional Vavoom engine Extensions

Vavoom engine has added some extensions to DECORATE, so that it can be combined with any additions implemented via progs, like the additional DECORATE definitions (vavoom_decorate_defs.xml), that allows authors to use any new defined actor properties for custom games.

DECORATE definitions

Basic concept

"**vavoom_decorate_defs.xml**" is a lump that has to be added to the root of basepak PK3 files that come with custom games, it's a simple XML file that contains extra definitions for any new flags and properties that come from custom games.

It uses a simple XML syntax described below.

Every XML file should begin with the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

They also should have an opening label like this one:

```
<vavoom_decorate_defs>
```

After this label, authors can define class properties and flags using the following definitions. When all definitions are finished, the file must end with the following label:

```
</vavoom_decorate_defs>
```

Properties Definition

All properties need to have 3 things: A **type**, a **name** for DECORATE and the name of the **property they represent in progs**, so for example we can define a property like Health for things like this:

```
<prop_int name="Health" property="Health" />
```

This syntax means that we are defining a property that will require integer numbers, named ″Health″ in DECORATE and that affects the ‘Health’ property in progs.

Also notice from the above syntax that property definitions should begin with a **<** and they should end with **/>**.

Here’s a list of available property types:

- prop_int: Defines a property that will require ″integer″ numbers.
- prop_float: Defines a property that will require ″decimal″ values.
- prop_name: Defines a property that will require a ″name″ type.
- prop_string: Defines a property that will require a ″string″ type (remember that strings should be between ″).
- prop_class: Defines a property that will require a ″class″.* ″’prop_state”’: Defines the name of a new state for the thing.

Flags Definition

Similarly, flags require 2 things: A ″’name”’ for DECORATE and the name of the ″’flag they represent in progs”’, we could define a flag like this:

```
<flag name="Solid" property="bSolid" />
```

This syntax example means that we are defining a property named ″Solid″ in DECORATE and that will affect the ″bSolid″ flag in progs.

Assigning Properties and Flags to Classes

To assign a property or flag to a certain class you should do something like this:

```
<class name="Actor">
  <prop_int name="Health" property="Health" />
  <prop_int name="GibHealth" property="GibsHealth" />
  <prop_int name="WoundHealth" property="WoundHealth" />
  <prop_int name="ReactionTime" property="ReactionCount" />
  <flag name="Solid" property="bSolid" />
  <flag name="Shootable" property="bShootable" />
  <flag name="Float" property="bFloat" />
  <flag name="NoGravity" property="bNoGravity" />
</class>
```

Notice that we have two new labels here: **<class name=”Name”>** indicates the name of the class the following properties and flags are defined for and **</class>** indicates the end of this class definition. You can use as many of these blocks as the authors wish, they just have to make sure that the class names are correctly defined in their DECORATE scripts.

Example

Here’s a small example of Vavoom definitions for DECORATE:

```
<?xml version="1.0" encoding="UTF-8" ?>
<vavoom_decorate_defs>
  <class name="HexenWeapon">
    <prop_class name="HexenWeapon.AmmoType3" property="AmmoType3" />
    <prop_int name="HexenWeapon.AmmoGive3" property="AmmoGive3" />
    <prop_int name="HexenWeapon.AmmoUse3" property="AmmoUse3" />
  </class>
</vavoom_decorate_defs>
```

Actor states

Actor states are described in more detail on the ZDOOM Wiki, but in a nutshell they are used to define the actions an actor can take along with their visual representation.

Linking actor states to sprites

When using sprites, the basic decorate function works like described in the ZDOOM Wiki, the first two elements defining what the actor looks like in that step. For example, “ETTN A” means that the sprite ETTNA0 is displayed during the Spawn state.

```
Spawn:
        ETTN AA 10 A_Look
        Loop
See:
        ETTN ABCD 5 A_Chase
        Loop
Pain:
        ETTN H 7 A_Pain
        Goto See
Melee:
        ETTN EF 6 A_FaceTarget
        ETTN G 8 A_CustomMeleeAttack(random(1, 8) * 2)
        Goto See
Death:
        ETTN IJ 4
        ETTN K 4 A_Scream
        ETTN L 4 A_NoBlocking
        ETTN M 4 A_QueueCorpse
        ETTN NOP 4
        ETTN Q -1
        Stop
```

Linking actor states to models

Decorate doesn’t know that you are using models, so it interprets the original sprite definition and passes this information to the rendering engine. If there is also a model definition for the Actor class, Vavoom checks the state index and if it then finds a matching state, it’ll display that 3D model frame instead of the sprite. If it doesn’t find a matching state in the model definition state index or if the model frame can’t be rendered for some reason (e.g. the MD2 file is missing), the renderer will fall back to the sprite and display that instead (to avoid this, you can define TNT1 as sprite, which is not rendered).

```
states
{
  Spawn:
      ETTN A 10 A_Look
      Loop
  See:
      ETTN ABCD 5 A_Chase
      Loop
  Pain:
```

```
        ETTN A 7 A_Pain
        Goto See
    Melee:
        ETTN EF 6 A_FaceTarget
        ETTN G 8 A_CustomMeleeAttack(random(1, 8) * 2)
        Goto See
    Death:
        ETTN IJ 4
        ETTN K 4 A_Scream
        ETTN L 4 A_NoBlocking
        ETTN M 4 A_QueueCorpse
        ETTN NOP 4
        ETTN Q -1
        Stop
}
```

Vavoom maps states from the model definition in a completely different way than it does with sprites: it parses the whole *states* list and whenever it finds a letter in the second column of an entry, it maps that to the next available model state number. It doesn't matter what the letter is and whether it is unique within the state or even the entry (that's why "ETTN AA" from the original sprite decorate definition has been changed to "ETTN A", otherwise the idle animation at spawn would alternate between the idle frame and the first walk frame, and all subsequent mappings would be off by one).

This means that the end result of the above Decorate definition has the following model mapping:

```
    Spawn:
        state 0
    See:
        state 1 state 2 state 3 state 4
    Pain:
        state 5
    Melee:
        state 6 state 7
        state 8
    Death:
        state 9 state 10
        state 11
        state 12
        state 13
        state 14 state 15 state 16
        state 17
```

Thus, making the following modifications to the Actor states results in no visible changes when using models:

```
states
{
    Spawn:
        ETTN A 10 A_Look
        Loop
    See:
        ETTN DJJL 5 A_Chase
        Loop
    Pain:
        ETTN A 7 A_Pain
        Goto See
    Melee:
        ETTN AA 6 A_FaceTarget
        ETTN A 8 A_CustomMeleeAttack(random(1, 8) * 2)
        Goto See
    Death:
        ETTN CW 4
        ETTN C 4 A_Scream
        ETTN D 4 A_NoBlocking
```

```
        ETTN E 4 A_QueueCorpse
        ETTN FGH 4
        ETTN I -1
        Stop
}
```

Basically, this means that you can add up to 29 independent animation steps for every actor state instead of the total 29 animations steps for the actor, as is the case with sprites (which also can be worked around by using more than one set of sprites for an actor).

While at first look it also means you can't use the same model frame in two different states, this isn't actually a problem as within the model definition file, you can map as many states to the same model frame as you want (also, the model frames used for the model states of an Actor can originate from the frames of any number of MD2 files).