Particle Effects in Halo Custom Edition - A Reference Guide

By Ifafudafi

(Thanks to Teh Lag for correcting a few things)

Contents

I. Introduction
II. Sprite basics
III. The .effect tag
IV. The .particle tag
V. The .particle_system tag
VI. The .decal tag
VII. Appendices

I. Introduction

Since CMT is quite literally the only thing still tethering me to Halo Custom Edition, I figured I'd better take the time to compile and record everything I've learned about its particle engine while I'm still working with it, especially considering the apparent dearth of decent tutorials and references on the subject. While I find it to be fairly intuitive and well-featured (at least compared to its contemporaries), there are enough inconsistencies, obscure in-jokes, and outright lies peppered around HCE's particle system (wait until you see the "Marty traded his kids for this" option) that I'm still regularly discovering new things I can and (more often) can't do even after nearly six years of effect artistry.

I should stress that this is indeed a "reference guide" rather than a straight-up tutorial; its objective is to explain and elaborate on the options and uses of each tag, as opposed to providing a step-by-step methodology. If you have even a cursory knowledge of how to operate Guerilla and Tool, though, it should be in-depth enough to give you everything you need to know to start designing your own effects. Getting things just the way you want them is less a matter of knowing the system and more a matter of constantly tweaking, polishing, and updating them until they're just right - the easiest way to mediocre work is giving in to the "oh well it's good enough for now" line of thought.

This guide will cover the **.effect**, **.particle**, and **.decal** tags in-depth, as they are the most crucial elements of good particle work. The **.particle_system** tag is less essential, there are so many tutorials on **.bitmap** tags that I don't feel I need to spend too much time with them, and **.point_physics** tags are simple and straightforward enough that explaining every detail shouldn't be necessary. The rest is down to practice - you'll probably want to start by simply copying and modifying existing effects (whether from vanilla Halo or from other mods) until you start to develop a "feel" for what timings, sizes, and such are ideal. While I can create an .effect from scratch, it is *incredibly* rare for it to ever be perfect on the first compile - if you're doing your job right, you'll be editing, saving, compiling, testing, and re-editing dozens of times over for nearly every single .effect.

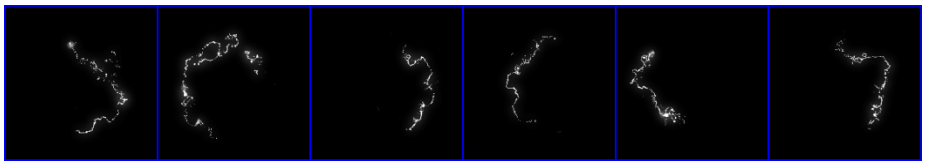
If you want a buttload of example effects, both of my largest projects - Another Stupid Campaign Mod and CMT SPv3 - are open-source; you can find download links on Halomaps, Modacity, and Halomods. The best way to learn is through personal experimentation - while I can spend thousands of words explaining what every bit of every tag does, you can grow much more quickly and comprehensively by playing around with those tags yourself. Wonder what changing a number does? Do it and find out! That's how I learned almost everything I know about HCE - tossing random ideas and values at Guerilla and Tool and seeing what happens. Hopefully, the information present in this reference guide will be enough to nudge you over most of the speed bumps I've had to go through, and will allow you to squeeze just a little more life out of this zombie of a modding community.

-Ifafudafi

II. Sprite basics

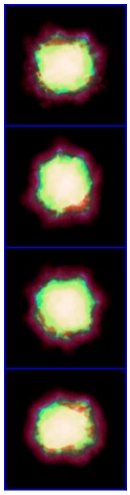
The most fundamental component of the particle effect is the **sprite**; .particles and .effects simply take a 2D bitmap and apply various properties. There are already tons of excellent tutorials on drawing sprites and getting them into tag form, so I'll assume you have some clue what you're doing and limit this section to detailing how to get a couple of specific results.

If you want your sprite to be animated, arrange your source .tiff horizontally, like so:



Halo will interpret this as **one sequence** with **six frames**.

If you want your sprite to be unanimated but to have several variations, arrange the source .tiff vertically:



Halo will interpret this as **four sequences** with **one frame** each.

You could also have multiple sequences each with multiple frames if you so desired.

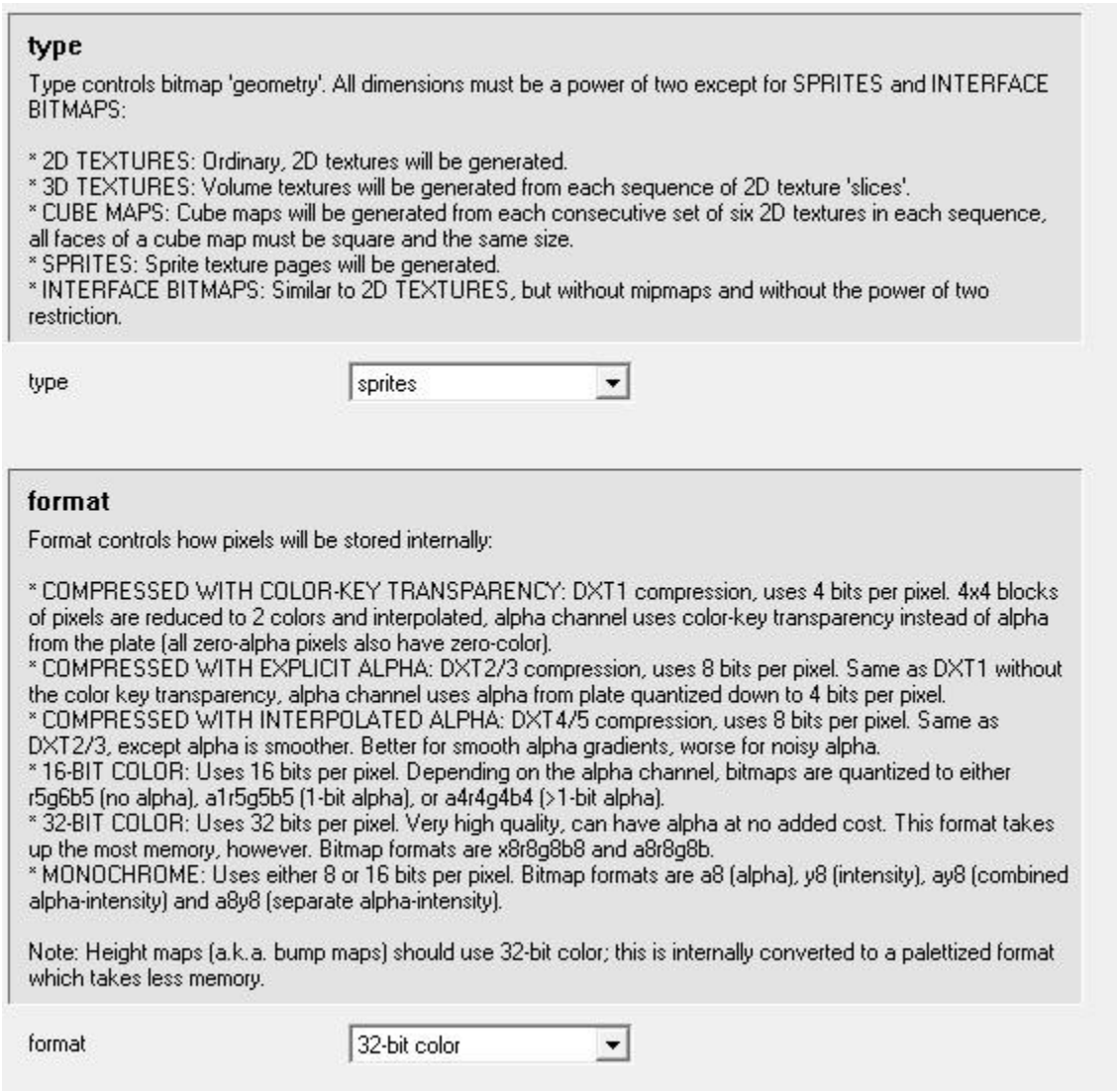
In all cases, you want each frame and/or sequence to be bordered by solid 0-0-255 blue - Halo interprets that color as a sprite's boundaries. It also means you can't use 0-0-255 blue in your sprites, but 0-0-254 is a near imperceptible difference, so it shouldn't cause any issues.

For sprites that are supposed to be "bright" or "glowy," such as sparks and muzzle flashes, you'll want to use solid black backgrounds with no alpha channel. For sprites that are supposed to be more "solid" or

"dull," such as wooden chips or dirt clouds, you'll want to use a transparent background. For decals that are supposed to be more "solid" or "dull," you'll want to use solid white backgrounds with no alpha channel. For more information on why, see Framebuffer blending.

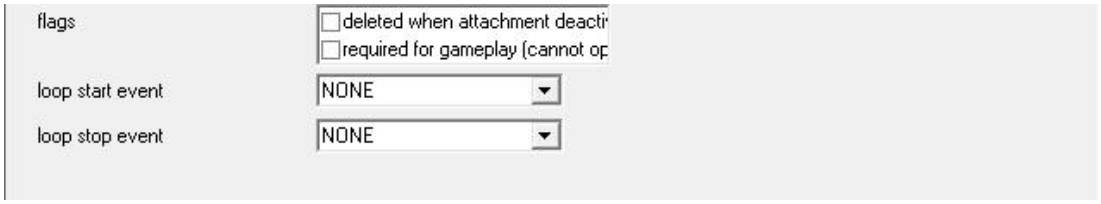
Don't forget to set your bitmap type to "sprites" in your bitmap tag, and, if you're using transparency, you *must* set your format to "compressed with explicit alpha" *at least*. Explicit alpha is lower-quality than interpolated alpha which is lower-quality than 16-bit color which is lower-quality than 32-bit color, but the increase in quality comes with an increase in tag size, especially when used for larger sprites (256x256 per frame/sequence or larger).

You can set a hard limit on sprite size and count in the .bitmap's "sprite processing" section, but it isn't necessary, especially for the purposes of making custom content. I would recommend you leave this "blank" (sprite budget size to 32x32 and sprite budget count to 0) unless you're actively trying to avoid hitting filesize limits in Tool.



III. The .effect tag

This will be the "core" of your particle effect; the .effect references every other tag utilized, whether it's a .particle or a .particle_system, a .sound or a .decal, or even a .projectile or .weapon. An .effect is usually attached to a .weapon, .biped, .projectile, or .vehicle, whether as an actual "Attachment" scaling itself using one of those tags' function blocks or a specific fire effect for a .weapon or impact effect for a .projectile. Here's a list, element by element, top to bottom, of what exactly it does:

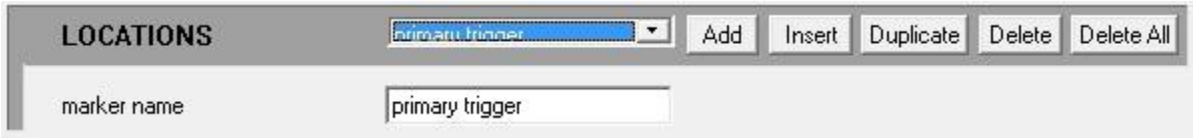


Flags
Deleted when attachment deactivated: If the effect is created as an attachment, delete the effect when the attachment is deactivated.
Required for gameplay (cannot optimize): The elements in this .effect's Parts blocks will always play, even if the .effect cap has been reached. Particles will still be dropped.

Loop start event: If this effect is attached to a looping function input (such as overheat or charging), begin the loop at this Event.

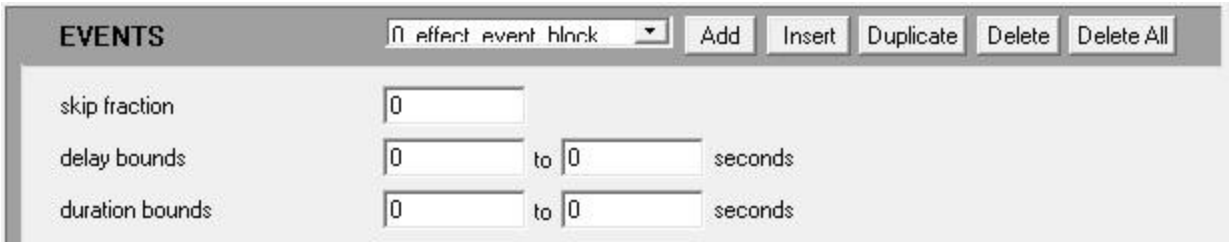
Loop stop event: If this effect is attached to a looping function input (such as overheat or charging), return to the Loop start at this Event.

Note: If the Loop start and Loop stop events are set to NONE, the first and last Events in the .effect will be treated as the Loop start and Loop stop events.



LOCATIONS: A list of markers at which this .effect's Parts and Particles can spawn.

Marker name: If the particle effect is spawned by attachments or firing effects, you'll probably want to use the markers attached to the corresponding model (such as "primary trigger" and "vent" on the Plasma Rifle). Alternatively, you can use one of a set of universal markers; these are best for .effects that spawn independent of a model, such as explosions and projectile impacts. See Universal markers for more information.



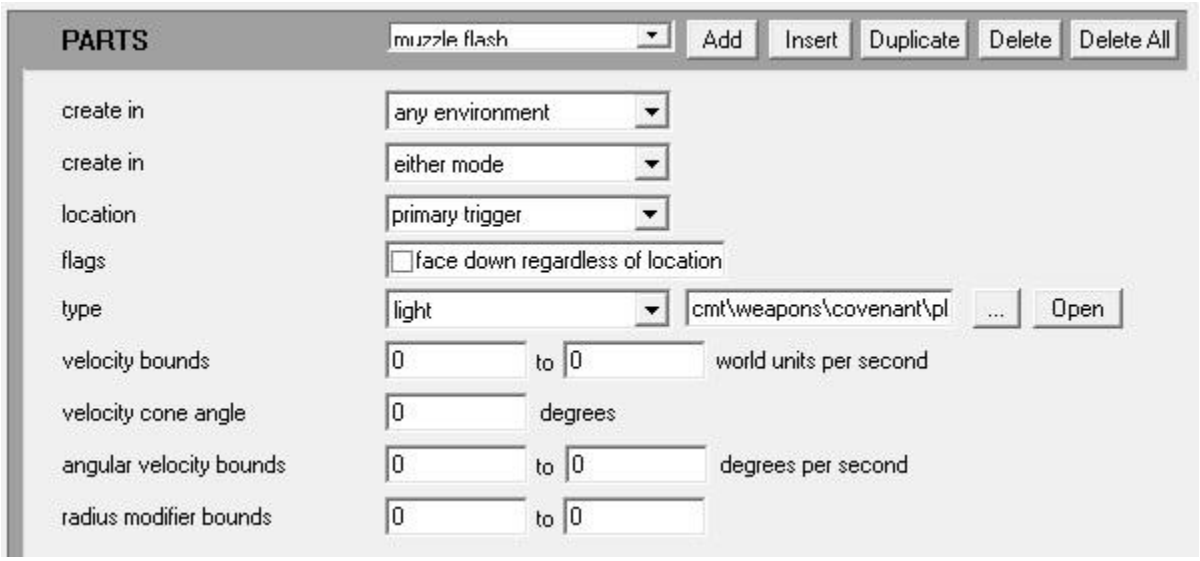
EVENTS: An .effect spawns Parts and Particles as part of an Event. The primary use of multiple Events is to separate stages of an effect at different points in time, but they can also be used to simply organize the .effect - putting first-person-only and third-person-only Particles in their own Events, for example.

Skip fraction: The chance this Event block will be skipped entirely. At its minimum value of 0, the Event will always occur. At its maximum value of 1, the Event will never occur. At, for example, a value of 0.3, the event has a 70% chance of occurring.

Delay bounds: The amount of time, in seconds, that must pass before this Event occurs. The .effect will randomly pick a value between the first and second numbers entered.

Duration bounds: The amount of time, in seconds, that an Event takes to occur. The .effect will randomly pick a value between the first and second numbers entered.

Note: The next Event block will not occur until *both* the Delay and Duration times have reached completion.



PARTS: Specifies components of an .effect other than .particles. This is the most versatile and practically useful component of the .effect, as it can spawn anything from .decals to .projectiles to .bipeds to .weapons.

Create in [environment]: Specifies whether this Part should draw in air only, water only, space only, or all three.

Create in [violent mode]: Specifies whether this Part should draw in violent mode only, nonviolent mode only, or both. See "Violent mode" for more information.

Location: Specifies which marker to spawn this Part at. The markers that can be selected are defined in the Locations block near the top of the tag.

Flags
Face down regardless of location: This Part will always be pointed downwards. This does not affect actual movement direction.

Type: Specifies the type and location of the tag to be used. Valid tag types are listed in the drop-down selection.

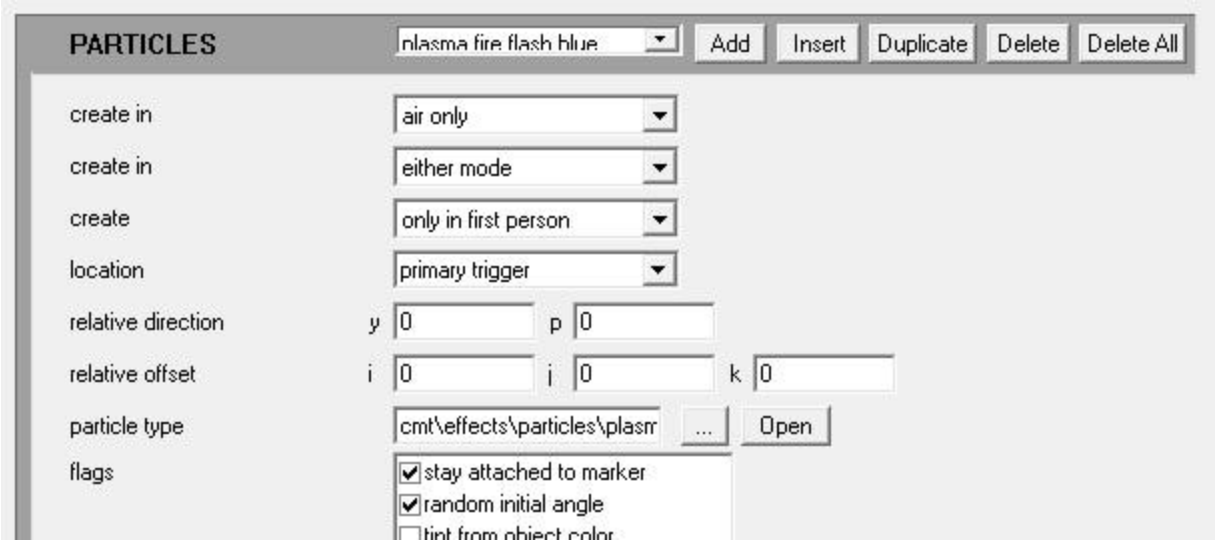
Velocity bounds: If applicable, the Part will spawn with this velocity, randomly selected between the first and second numbers. If the tag used already has a set velocity (such as a .projectile), the tag's velocity will be *multiplied* by the velocity specified here. If the tag is a .decal, this will specify how far from the .effect's spawn point the .decal can be created.

Velocity cone angle: Specifies the possible variation in this Part's direction. A direction will be randomly selected inside the specified cone size. See "Velocity cone diagrams" for more information.

Angular velocity bounds: Specifies the Part's rate of rotation, in degrees. A rotation rate will be randomly selected between the first and second numbers.

Radius modifier bounds: If applicable, the Part's radius will be *multiplied* by a randomly selected value between the first and second numbers.

Scale modifiers: See Scale modifiers for more information.



PARTICLES: Particles. A .particle is referenced, and then is run through all the flags and modifiers set here.

Create in [environment]: Specifies whether the Particle should draw in air only, water only, space only, or all three.

Create in [violent mode]: Specifies whether the Particle should draw in violent mode only, nonviolent mode only, or both. See Violent mode for more information.

Create [camera mode]: Specifies whether the Particle should draw relative to the first-person model, third-person model, both, or neither. More specifically:
Independent of camera mode: The Particle will draw relative to the third-person model, but will also be visible in first-person. For most effects, this is the ideal setting, but because the third-person and first-person model are not aligned with each other, this will result in odd behavior if used, for example, on a weapon fire effect.
Only in first person: The Particle will draw relative to the first-person model, and it won't be visible in third-person. Select this for effects attached to a first-person model, such as FP arms or weapons.
Only in third person: The Particle will draw relative to the third-person model, and it won't be visible in first-person. This will allow you to, for example, make a weapon's firing effect larger (and thus more visible) in third-person but smaller (and more suited to the weapon model) in first-person.
In first person if possible: If the effect is attached to something with a first-person model (such as FP arms or weapons), it will draw and be visible in first-person, but will also draw and be visible in third-person. This is the most convenient setting, but requires both first-person and third-person to share identical settings.

Location: Specifies which marker to spawn the Particle at. The markers that can be selected are defined in the Locations block near the top of the tag.

Relative direction: Modifies the Particle's direction, using the selected marker location as a baseline. First field is yaw (horizontal rotation), second field is pitch (vertical rotation). Uses degrees (0 to 360), rotating clockwise.

Relative offset: Modifies the Particle's spawn location, using the selected marker location as a baseline. Uses world units. "i" is the forward-to-backward axis (positive values move forward, negative values move backward), "j" is the left-to-right axis (positive values move left, negative values move right), "k" is the upward-to-downward axis (positive values move upward, negative values move downward). This setting *does not* take a relative direction setting into account.

Particle type: Specifies the .particle tag to use.

Flags

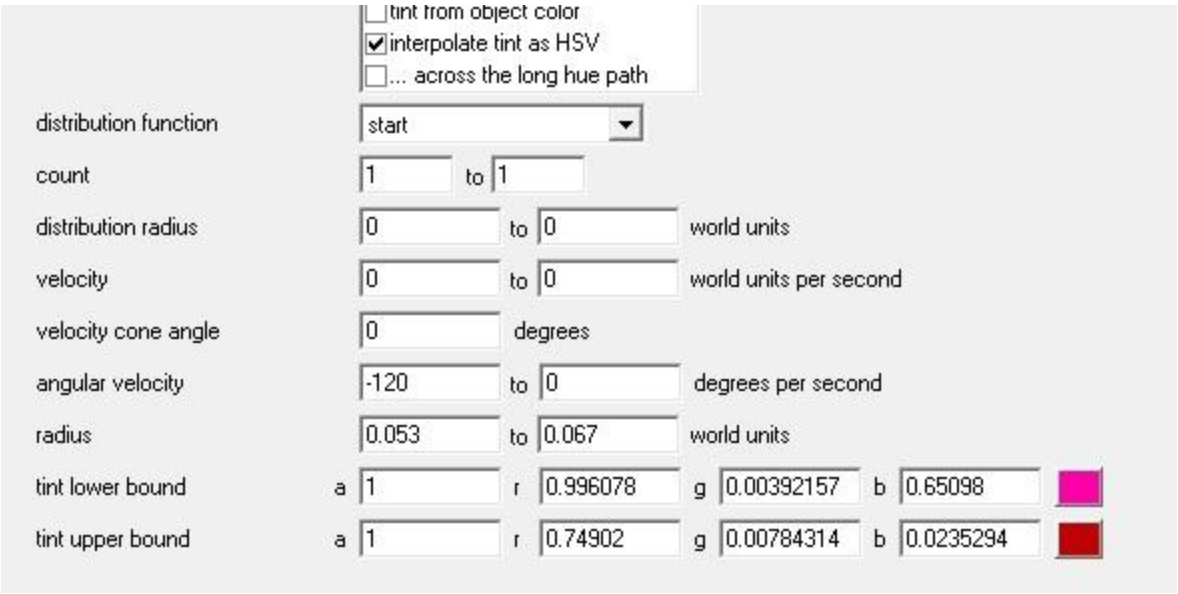
Stay attached to marker: The Particle will remain attached to the selected location marker. Velocity and angular velocity will be ignored if this flag is checked.

Random initial angle: The Particle will spawn at a randomly selected angle.

Tint from object color: The Particle's tint will be inherited from the color of the object it is spawned from - if the .effect is an Attachment and the Attachment has a change-color selected, that will be the color used. The inherited color will *multiply* the tint you give the Particle.

Interpolate tint as HSV: Particle's tint will be computed on a Hue-Saturation-Value scale rather than a Red-Green-Blue scale. If the particle's tint range is very large (ex. red to green), this may prevent glitches and other oddities.

…across the long hue path: If "interpolate tint as HSV" is selected, this Particle will take the longest possible route between its hue range. For example, if 0-0-1 blue and 1-0-1 pink are the selected tint ranges, the possible color variations will go from blue to cyan, green, yellow, and red before reaching pink, rather than going directly to pink.



Distribution function: Specifies how the Particle's "count" value should be utilized relative to the Event block's duration timer.
Start: All Particle instances will be drawn simultaneously at the beginning of this Event block's duration timer.
End: All Particle instances will be drawn simultaneously at the end of this Event block's duration timer.
Constant: Particle instances will be spread evenly along this Event block's duration. For example, if the Event's duration is 2 to 2 seconds and the Particle's count is 4 to 4, a new particle instance will draw every 0.5 seconds.
Buildup: Particle instances will spawn at a slow rate at the beginning of the Event's duration timer, but will gradually increase in spawn rate as the duration time passes.
Falloff: Particle instances will spawn at a fast rate at the beginning of the Event's duration timer, but will gradually decrease in spawn rate as the duration time passes.
Buildup and falloff: Particle instances will spawn at a slow rate at the beginning of the Event's duration timer, increase in spawn rate until the duration timer is *halfway complete*, and then decrease in spawn rate as the timer finishes.

Count: Specifies how many particle instances to draw. A value will be randomly selected between the first and second numbers entered.

Distribution radius: Specifies an area in which particle instances will spawn. If set to 0-0, particle instances will always draw in the same location; otherwise, an area will be randomly selected between the first and second values, and each particle instance will draw at a random location within this area.

Velocity: Specifies the Particle's velocity. A value will be randomly selected between the first and second numbers.

Velocity cone angle: Specifies the possible variation in particle instance direction. A direction will be randomly selected inside the specified cone size. See "Velocity cone diagrams" for more information.

Angular velocity: Specifies the potential rates of rotation for each particle instance. A rate of rotation will be randomly selected between the first and second numbers.

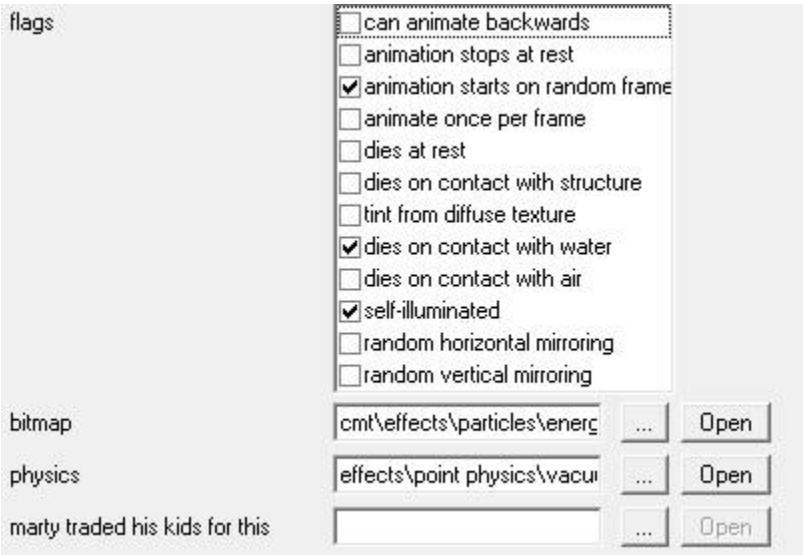
Radius: Specifies the potential size for each particle instance. A radius will be randomly selected between the first and second numbers.

Tint: Specifies the potential transparency and color of each particle instance, using an Alpha-Red-Green-Blue scale. A value will be randomly selected between the lower and upper bounds. Clicking on the color preview to the right will open up a color selection window.

Scale modifiers: See Scale modifiers for more information.

IV. The .particle tag

Anything not defined in the .effect is defined in the .particle. Depending on how complex your effects are, it is quite possible you may have multiple .particles sharing the same bitmap so that you can have different variations on animation rates, lifetimes, and such. A .particle is fairly negligible when it comes to tagspace & filesize, so don't be afraid to create as many as you need.



Flags

Can animate backwards: The .particle has a random chance to animate backwards (i.e. its last frame becomes its first frame and vice versa).

Animation stops at rest: If the .particle loses all velocity, its animation will stop.

Animation starts on random frame: The .particle's animation starts on a random frame. This can be combined with "can animate backwards."

Animate once per frame: The animation rate will be interpreted as *full animations per second* rather than *frames per second*. This is not a very useful setting, as the same thing can be done with a higher frames-per-second value.

Dies at rest: If the .particle loses all velocity, it will delete itself.

Dies on contact with structure: If the .particle comes into contact with the BSP, it will delete itself. The particle's .point_physics must enable collision with structures for this flag to take effect.

Tint from diffuse texture: The .particle will tint itself based on the diffuse texture of the .bitmap. The same effect can be achieved by selecting a tint of 1-1-1 in the .effect, rendering this flag more or less useless.

Dies on contact with water: If the .particle comes into contact with water, it will delete itself. The .particle's .point_physics must enable collision with water surfaces for this flag to take effect.

Dies on contact with air: If the .particle comes into contact with air (i.e. is not in water or in space), it will delete itself.

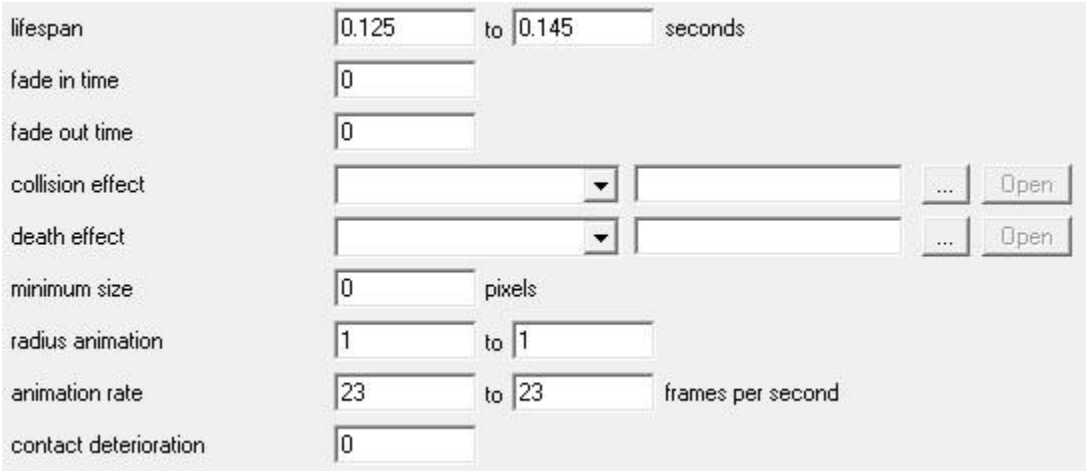
Self-illuminated: The .particle will always draw at full brightness, ignoring any world or dynamic lighting.

Random horizontal mirroring: The .particle has a random chance to flip itself horizontally.
Random vertical mirroring: The .particle has a random chance to flip itself vertically.

Bitmap: Specifies the .bitmap tag to use. The .bitmap type *must* be "sprites."

Physics: Specifies the .point_physics tag to use.

Marty traded his kids for this: Specifies a .sound to play upon particle collision. This can be accomplished by using a .sound or .effect in the "collision effect" field, making this setting more or less useless.



Lifespan: Specifies how long this .particle will exist. A value will be randomly selected between the first and second number.

Fade in time: Specifies how long this .particle takes to reach its maximum alpha value (specified in its .effect).

Fade out time: Specifies how long this .particle takes to become fully transparent (alpha value at 0).

Collision effect: If the .particle can collide with a structure or water surface (specified in its .point_physics tag), play this .effect or .sound on collision.

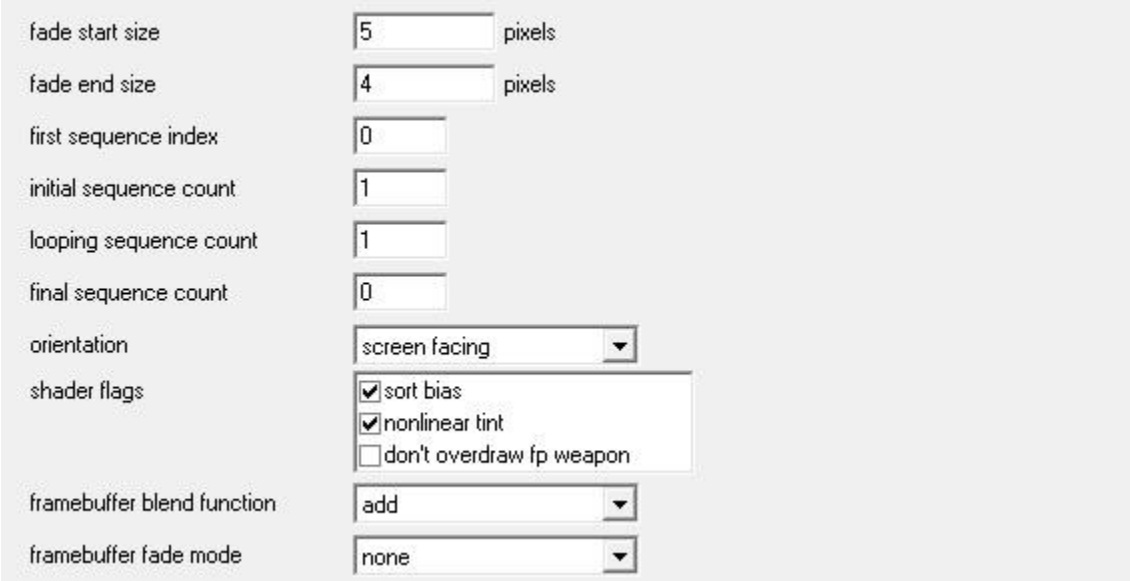
Death effect: When the .particle dies, play this .effect or .sound.

Minimum size: The .particle will never shrink below this size, no matter how far away it is from the player.

Radius animation: The .particle will increase or decrease in size across its lifespan, beginning at the size specified by the first number and ending at the size specified by the second number. These values *multiply* the radius given to the particle in its .effect.

Animation rate: Specifies the rate of animation, if the .particle's .bitmap is set up to support animation. If all frames have been drawn before the particle's lifespan expires, the animation will loop from the beginning.

Contact deterioration: If the .particle can collide with structures or water surfaces, the animation rate will be lowered by this many frames per second if it does collide.



Fade start size: If the .particle's size on the screen is less than this, it will grow more transparent until it reaches "fade end size" below.

Fade end size: If the .particle's size on the screen is equal to or less than this, it will be fully transparent.

First sequence index: Specifies which sequence in the .bitmap to treat as the first sequence (for either selecting random sequences or for animation).

Initial sequence count

Looping sequence count

Final sequence count: These three settings do not appear to do anything significant. You can leave them as illustrated above or play around and see if you can figure out something I can't.

Orientation: Specifies how the .particle should be drawn relative to its direction.
Screen facing: The .particle will always be facing the player.
Parallel to direction: The .particle will face the direction specified in its .effect.
Perpendicular to direction: The .particle will face 90 degrees (right angle) away from the direction specified in its .effect.

Shader Flags
Sort bias: The .particle will always draw on top of .particles which do not have their sort bias flag checked.
Nonlinear tint: The higher the value (i.e. Hue-Saturation-Value scale) of the .particle's .bitmap, the less its .effect's tint will apply. Pixels that are at the maximum value (i.e. white) will not be tinted, while pixels that are at the minimum value (i.e. black) will be fully tinted.
Don't overdraw fp weapon: This .particle is not allowed to draw over a first-person model.

Framebuffer blend function: Specifies how to interpret the .bitmap's transparency. See Framebuffer blending for more information.

Framebuffer fade mode
None: The .particle behaves normally.
Fade when perpendicular: The .particle will be fully visible when the player is directly facing its front or back, but will fade out if viewed from a different angle, becoming fully transparent when viewed from the side.
Fade when parallel: The .particle will be fully visible when the player is facing it from the side, but will fade out if viewed from a different angle, becoming fully transparent when viewed directly from the front or back.

V. The .particle_system tag

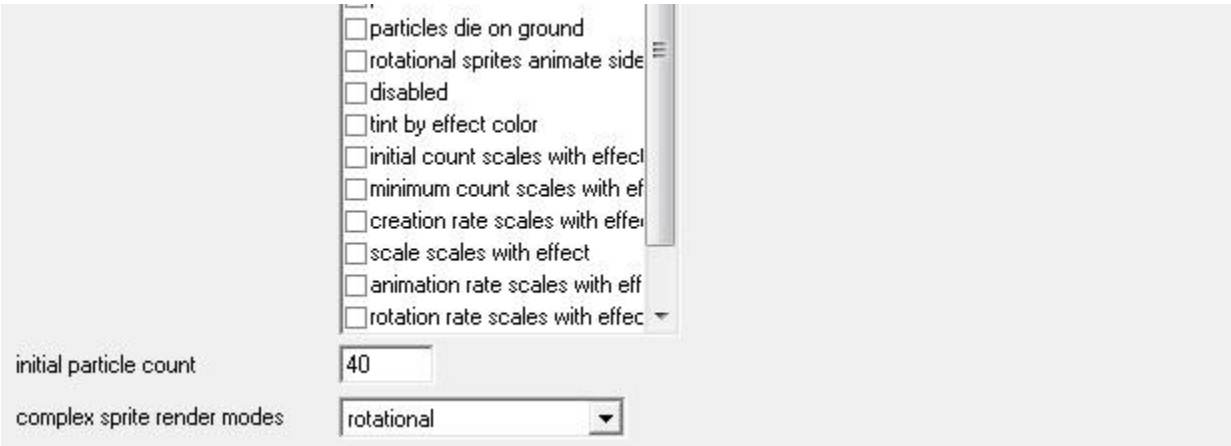
While ostensibly more versatile than .particles, .particle_systems are actually convoluted, unintuitive, buggy, cumbersome, facetious piles of ass-poo. The only good reasons for using one are:

1. A single particle instance can switch between multiple sprite bitmaps
2. You have finer control over the emitter's shape
3. Particles generated by a .particle_system are governed by a separate draw cap

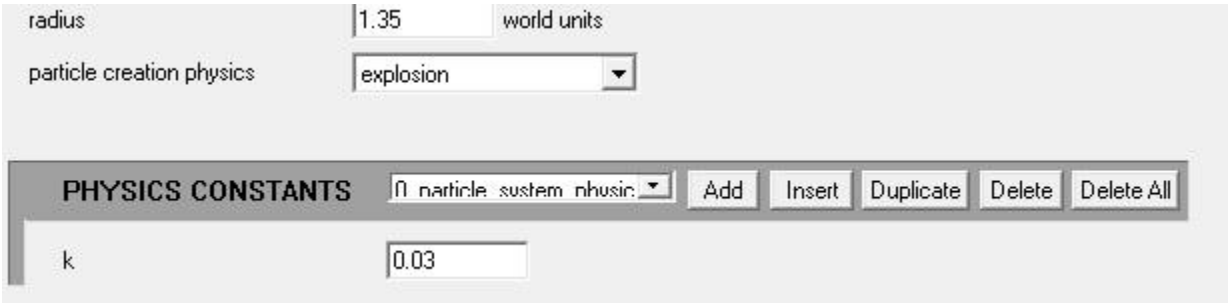
If your effect doesn't need any of those qualities, there is no reason to use a .particle_system. Unlike a .particle, particles generated by a .particle_system cannot:

1. Rotate
2. Self-illuminate
3. Create an .effect upon collision
4. Mirror themselves horizontally or vertically
5. Remain attached to a marker
6. Reliably draw in first-person
7. Scale their emission rate non-linearly (i.e. buildup/falloff)
8. Utilize violent mode controls

And more. For 90% of particle effects, you'll never have to use this tag; but for some things - especially explosions - you're just going to need that finer level of control, despite all the hoops you'll have to jump through. Because most of the fields in the .particle_system are irrelevant or non-functional, I'll cover this in a more directional manner.



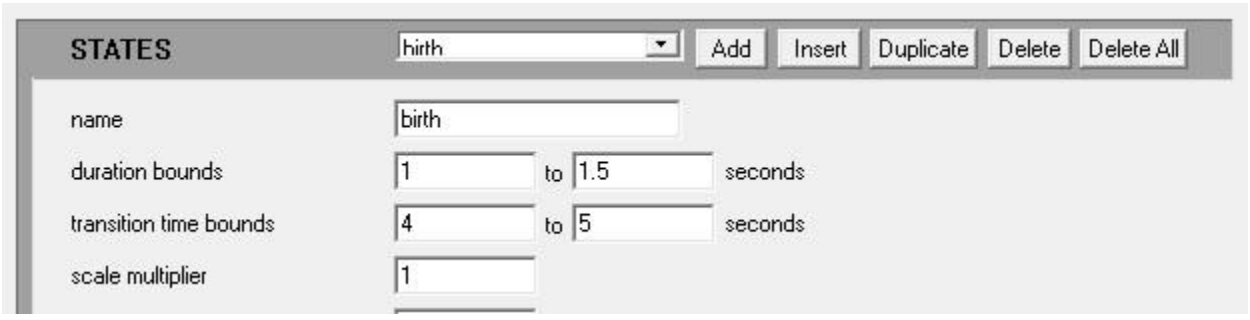
You can ignore everything up to the big list of flags; most of those are obvious enough. "…scales with effect" means that if the .effect is attached to a function, the .particle_system will scale the indicated properties along with that function (similar to Scale modifiers in an .effect). Particle count is self-explanatory, and "complex sprite render modes" has no visible effect that I can perceive.

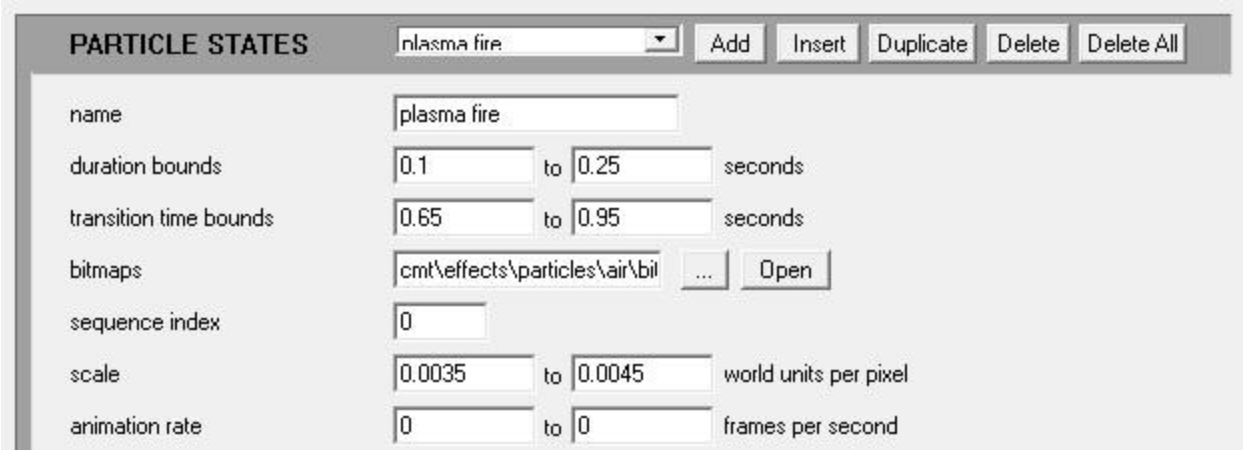


"Radius" is going to work in tandem with your "particle creation physics." A larger radius, intuitively, means that the particles extend farther, but precisely how far they extend depends on another couple of settings.

Here's what each "particle creation physics" setting does:
Default: The first physics constant scales overall radius; effective values should be in the hundreds. The first constant is the only one that should affect anything.
Jet: The first physics constant appears to scale overall radius; effective values should be in the hundreds. The second and third physics constants also seem to scale overall radius, but effective values for those two should be between 0 and 1.
Explosion: The first physics constant scales *horizontal radius* and the second constant scales *vertical radius;* in both cases, effective values should be between 0 and 1. The third constant scales overall radius; its effective values should be in the hundreds.

Exactly how large/small these values need to be will depend on your "radius" setting above. Personally, I can't see any reason to use anything but "explosion," but there may be some subtle differences I'm not able to notice through trial-and-error experimentation.





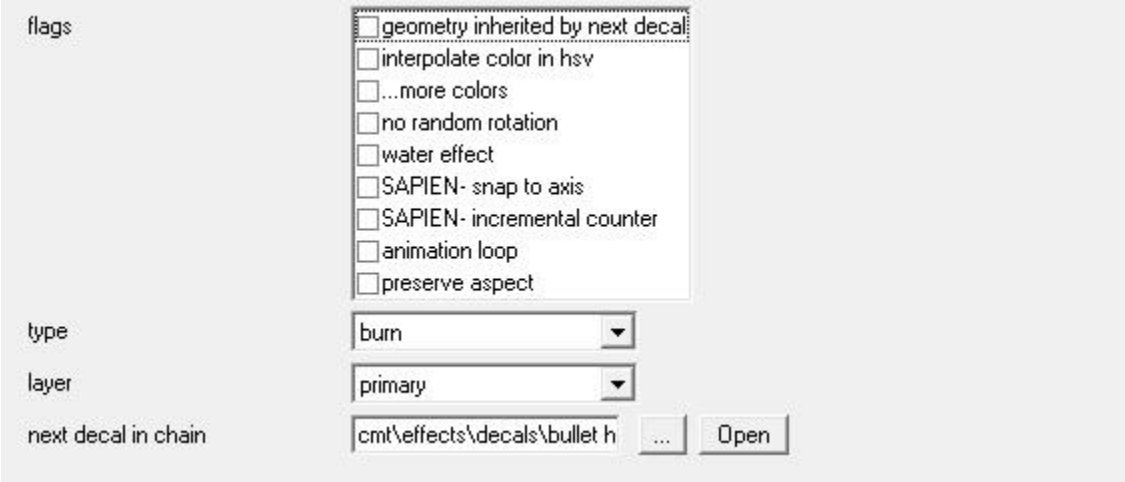
"States" and "Particle States" are the two most important elements of the tag. The former controls the *whole emitter*, while the latter defines the behavior of *individual particles*. Both act simultaneously, so you will need to make sure the duration and transition times line up. Here's some clarification, as a few fields are misleading at best and outright wrong at worst:

-Scale multiplier in States affects *particle size*, while **radius multiplier** in both States and Particle States affects *the .particle_system's size* (as defined under Particle Types)
-Both **Rotation rate multiplier** in States and **Rotation rate** in Particle States *do not work*. They instead seem somehow tied to animation rate, although I've never been able to figure out precisely how.
-**Animation_rate_multiplier** in States and **Animation rate** in Particle States do not seem to function properly on their own. I haven't figured out exactly how you have to work them, so the best I can recommend is play around with the numbers until it works for you.
-**Particle creation rate** spawns particles *in addition* to those immediately created (as defined by "initial particle count" under Particle Types). The particle creation physics and physics constants control these additional particles.
-Pay attention to the fact that **scale** under Particle States is in world units *per pixel*, rather than per sprite. This means that if you're transitioning from a lower-resolution sprite (ex. 64x64) to a higher-resolution sprite (ex. 128x128), you will get some visual oddities. In almost all cases, you'll want to avoid this by making sure all particle bitmaps in the .particle_system are the same resolution.

All other fields are either self-explanatory or function identically to their counterparts in the .particle tag.

VI. The .decal tag

Decals will usually be an essential component for explosions and projectile impacts. Fortunately, many of their characteristics are similar to .particles, as they both use "sprites" as their .bitmap type. You will almost always want to make use of the "multiply" framebuffer blending method rather than "alpha blend" for non-luminescent decals like bullet holes and Elite blood, however. Here's what you need to know:



Flags

Geometry inherited by next decal: If another .decal is referenced in "next decal in chain," it will inherit this .decal's radius, color, rotation, .bitmap sequence, and intensity. There are *very* few occasions where this will be useful, as you'll have to line relevant .bitmaps up with almost perfect precision.

Interpolate color in HSV: The .decal's color will be computed on a Hue-Saturation-Value scale rather than a Red-Green-Blue scale. If the .decal's color range is very large (ex. red to green), this may prevent glitches and other oddities.

…more colors: If "interpolate color in HSV" is selected, this .decal will take the longest possible route between its hue range. For example, if 0-0-1 blue and 1-0-1 pink are the selected color ranges, the possible color variations will go from blue to cyan, green, yellow, and red before reaching pink, rather than going directly to pink.

No random rotation: The .decal will always draw at the same angle.

Water effect: Unused, as far as I know. It doesn't seem to change anything when active, but leave it unchecked just in case there's some obscure error somewhere.

Animation loop: Animated .decals don't work, so this shouldn't affect anything.

Preserve aspect: Unused, as far as I know.

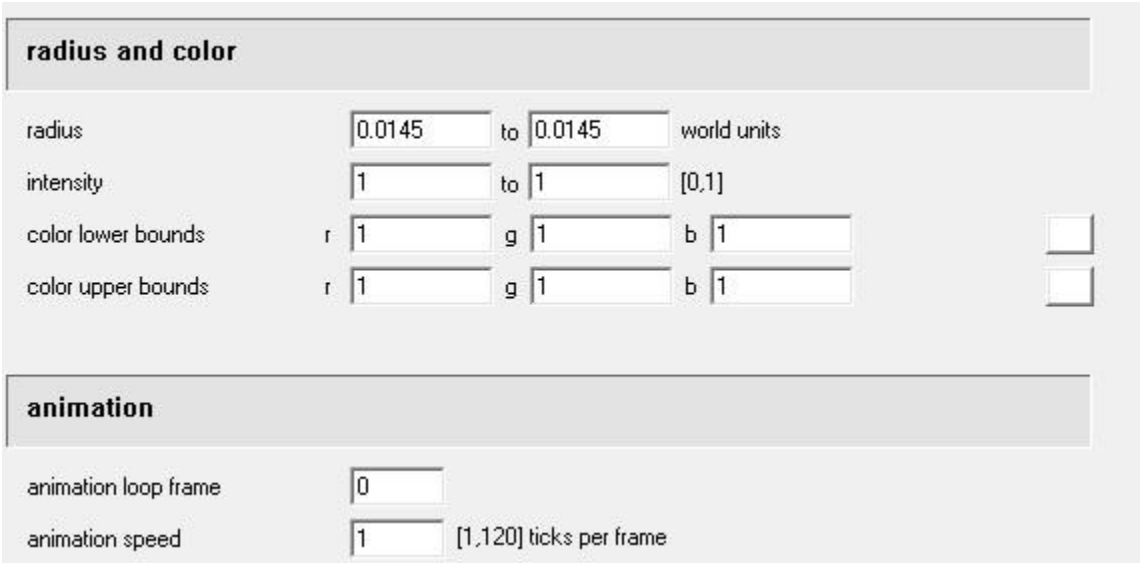
Type: Unused, as far as I know.

Layer

Primary: The .decal will draw normally.

Secondary: The .decal will draw normally, on top of decals in the primary layer.
Light: The .decal is drawn before the world texture. Practically, this means that the more well-lit the environment, the less visible a .decal in this layer is.
Alpha-tested/water: There is no need to use either of these layers.

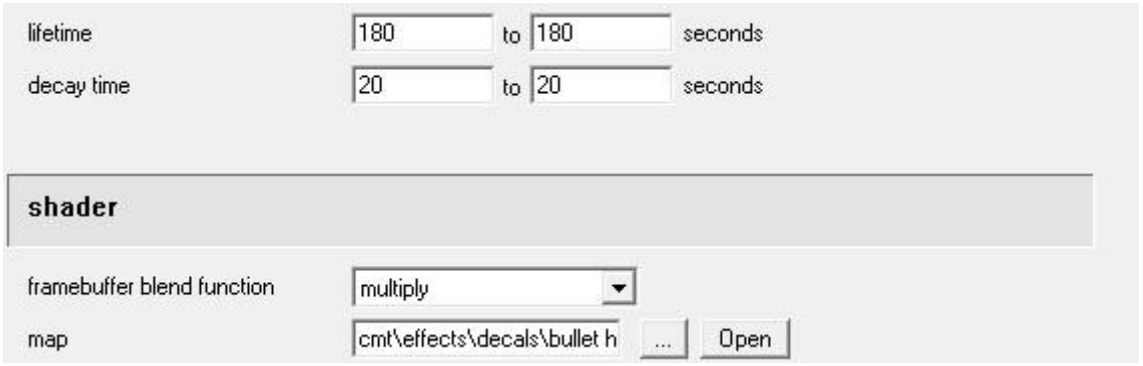
Next decal in chain: When this .decal is created, it will spawn the .decal referenced here in the same location. This is used in H1's plasma weapon impacts, for example.



Radius: This will *scale up with the .bitmap's resolution*. As best as I can tell, a 64x64 .bitmap will draw at 1 times the radius, a 128x128 .bitmap will draw at 2 times the radius, a 32x32 .bitmap will draw at half the radius, etc.

The next couple are self-explanatory, until **animation loop frame** and **animation speed**. Unfortunately, .decals cannot actually animate, so leave these at 0 and 1, respectively.

For the rest, see Framebuffer blending for information on how that works, and **map** is simply the .bitmap to use for this .decal.



VII. Appendices

Universal markers

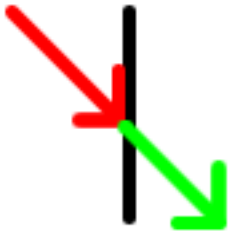
For effects that aren't attached to a model (like projectile impacts), you'll need to use one of these five markers; most of them (normal, reflection, gravity, negative incident) are designed specifically for that kind of thing. I've given each description an accompanying diagram.

If you leave the marker name blank: if the .effect is an Attachment, it will spawn at the origin of the object it is attached to. In all other cases, it will act as if you were using "normal" as your marker.

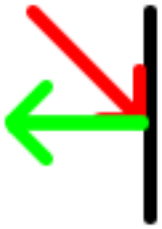
(**Red** is the initial direction, **Black** is the point where the .effect spawns, **Green** is the direction the marker represents)

incident: Preserves the angle of impact, but flips the direction.

negative incident: Continues along the angle of impact.



normal: Perpendicular to the impacted surface.

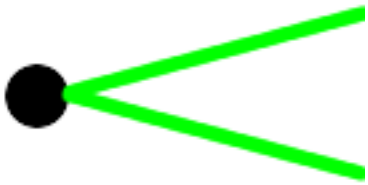


reflection: Pretty intuitive.

gravity: Angles in the direction of gravity.

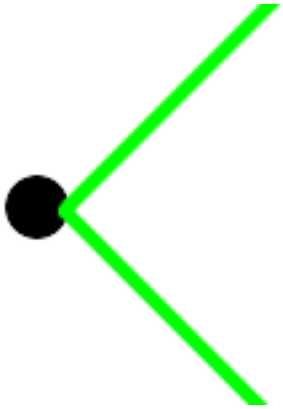
Velocity cone diagrams

Here are a couple of diagrams in case the text description isn't enough. The **black** sphere is the .effect spawn point, the **green** cone is the velocity cone.



This cone is about 30 degrees. A particle type whose velocity cone angle is set at 30 will spawn in a random direction *inside the green cone*. It could be a straight line, it could be at the very edge, it could be only a degree or two off the center.

This cone is set at 90 degrees:



The maximum angle is 360 degrees. If the velocity cone angle is set at 360, a particle could spawn in any direction; this is good for explosions, where you might want debris particles to fly out every direction.

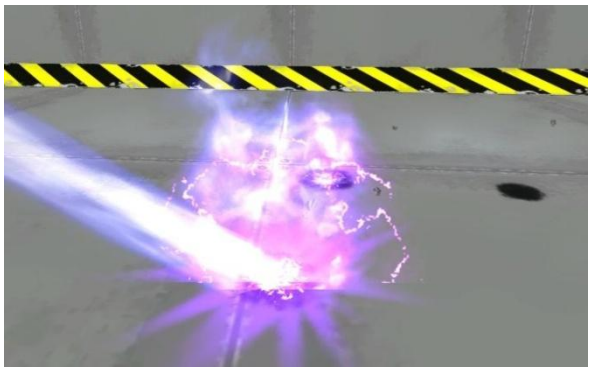
Framebuffer blending

Both .decals and .particles/.particle_systems use similar methods of interpreting transparency based on their .bitmaps.

Alpha blend: The .bitmap's alpha channel determines transparency; black is fully transparent, white is fully visible. Once transparency is computed, the sprite will be drawn "on top" of the frame. Along with add, this is usually the way to go with particles.

(These sand and dust particles use the **alpha blend** method, making them appear more "tangible")

Add: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - black is fully transparent, white is fully visible. Once transparency is computed, the remaining colors will be added to the frame's RGB values. Along with alpha blend, this is usually the way to go with particles, and should also be used for glowing decals (like plasma impacts).



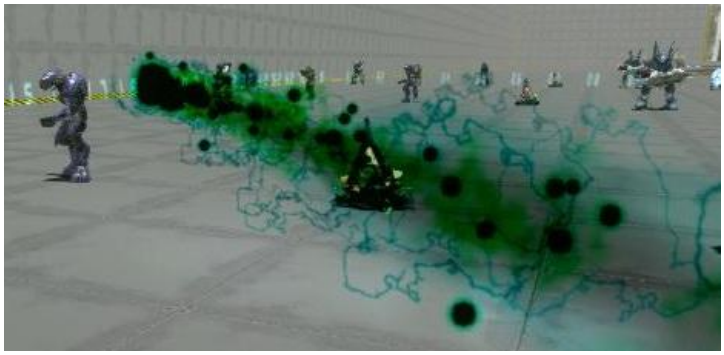
(These energy particles use the **add** method, making them appear luminescent)

Multiply: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - white is fully transparent, black is fully visible. Once transparency is computed, this will multiply the current frame's RGB values by those in the .bitmap. This is best used for scratch and liquid decals, like Elite blood or bullet impacts, but *does not work* for .particles and .particle_systems.

(These decals use the **multiply** method, ensuring that they're non-luminescent)

Double multiply: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - white is fully transparent, black is fully visible. It is supposed to double the brightness of the affected pixels before they are multiplied (so it doesn't look as dark) although I'm not sure about the precise operation. I'm not aware of any vanilla Halo sprite that uses this setting - I certainly don't.

Subtract: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - black is fully transparent, white is fully visible. This setting - quite aptly - removes the .bitmap's RGB values from the frame instead of adding them, creating a very unique but very circumstantial effect.

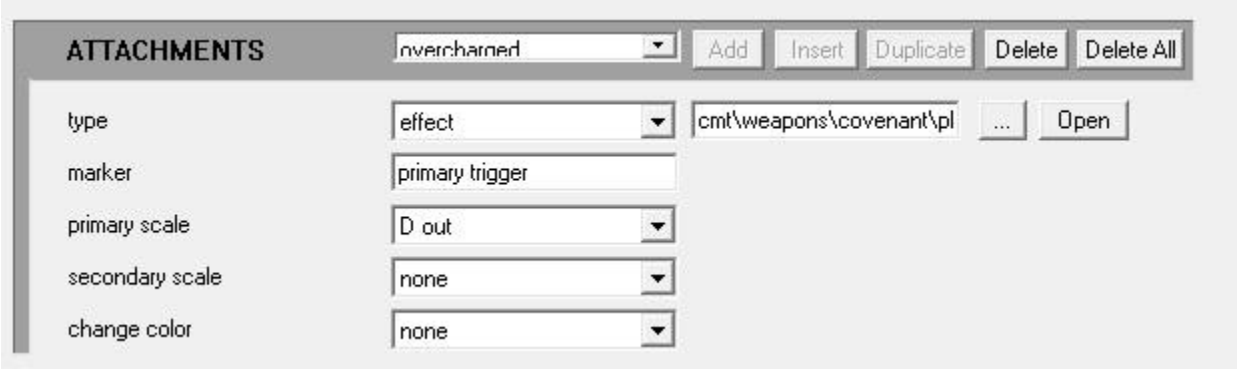


(These particles use the **subtract** method, making them appear as if they're "sucking the color out")

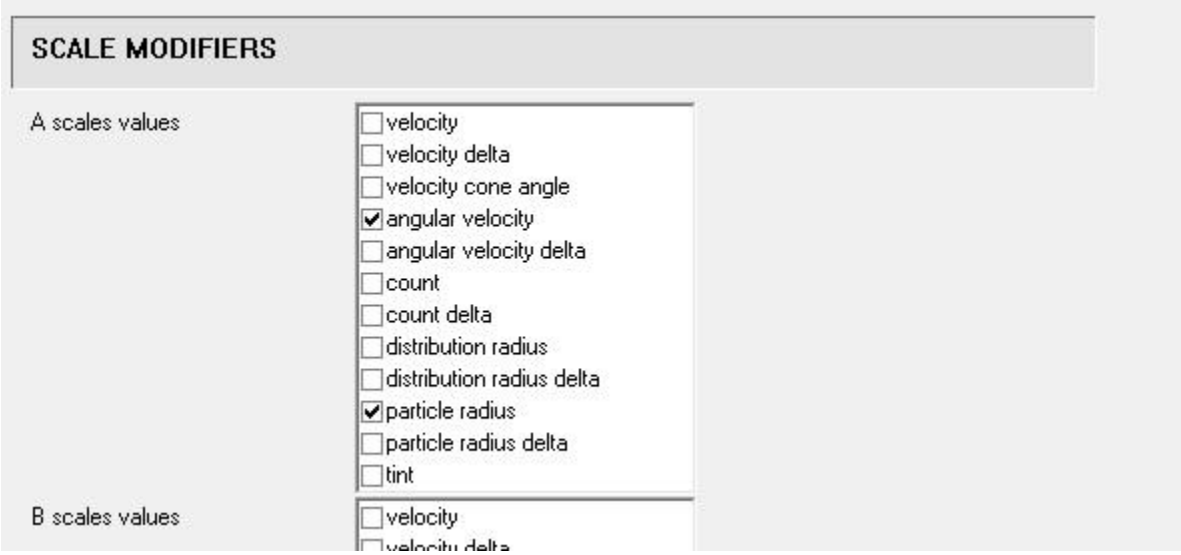
Component min: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - white is fully transparent, black is fully visible. I am not sure what this exactly this does on a technical level.
Component max: The value (HSV scale) in the .bitmap's diffuse channel determines transparency - black is fully transparent, white is fully visible. I am not sure what this exactly this does on a technical level. Grunt and Hunter blood decals use this setting.
Alpha-multiply add: Multiplies the .bitmap's diffuse RGB by its own alpha channel and then functions the same way as add. I don't know of any practical uses for this setting.

Scale modifiers

If you're creating an .effect as an Attachment, the .weapon or .biped or whatever will allow you to scale .effect elements by two of that object's functions. For example:



In the Plasma Pistol's .weapon, I attach the overcharge .effect to a function scaled by the gun's charge level. In the .effect, the function you set "primary scale" to will be treated as "A," and the one you set "secondary scale" to will be treated as "B":



Since I've checked these flags for one of the charging .effect's Particles, that Particle's angular velocity and radius will be *multiplied* by whatever the charging function's value is. If the gun is halfway charged, the particle will have 0.5 times (i.e. half) its normal radius, for example.

"Delta" is the difference between the lower and upper bounds of an element. If the Particle's radius was 0.4 to 0.8, for example, but I'd checked "particle radius delta" and the gun was halfway charged, the values would only range from 0.4 to 0.6.

Violent mode

When an .effect is spawned from a dead corpse (such as shooting or meleeing a dead Elite), Halo checks to make sure the relevant Particles and Parts are not "violent mode only." Particles or Parts that are violent mode only will *not* draw if spawned from a dead corpse. For example, this is used in CMT SPv3 to improve performance; .decals and Particles that create .decals are set to violent mode only to cut down on .decal count.

You can disable this check by typing "effects_corpse_nonviolent false" into the in-game console; all particles will then draw regardless of whether they are violent mode only or not.