GameCave Effects Engine 3.x

Particle Effects Tutorial

Written by Rhys Andrews

Table of Contents (Click to view chapter)

INTRODUCTORY

Particle Effects are one of the most well known visual/special effect methods used in GML, and in many cases, other languages as well. Particles are various shapes (and sizes), with very little information. This information includes where they're supposed to go, their appearance, and more. With such little information, particles take little CPU usage to calculate where and how to draw them on the screen. This is why particles are a great method to use if you want effects such as rain, explosions, smoke, fireworks, flames, and more. In the *GameCave Effects Engine*, many engines are created on particles alone (or sometimes with some non-particle effects attached). Unfortunately, Particle Systems/effects can take some time to get the hang of. They also take a lot of trial and error, and patience to get individual particle types the way you want them. Well, one alternative would be to use the *PartArt* implementation, included with the GCE3.X package. There are many other particle-designer programs you can download if you wish, however *PartArt* is the first and original.

On the other hand, you can get much more control and contentment through creating the particles via GML. This is where this tutorial comes in: to help you learn how particles work. Not just the particle-types, but their systems, emitters, destroyers, deflectors, and more. If you ever get stuck with certain functions and don't want to have to read this over again, you can always use the GM Manual (pressing F1 in the Game Maker's window). Other than that, enjoy the tutorial!

GETTING STARTED WITH PARTICLE SYSTEMS

Let's go through the basics of how a particle-system structure works. Every particle *system* contains groups of 'filters' like *emitters, destroyers, deflectors,* and *attractors*. Whether that made sense or not, the fact is, a system will hold a group of filters, and then according to the filters, displays the various particle types onto the screen in the correct way. Now you ask (or not), how do I create a particle system?

Particle Systems are created very easily.

```
system = part_system_create();
```

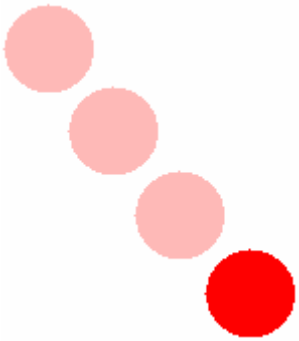
A new system has been created into the games memory. Now, whenever you need to edit, destroy, or insert filters/types into the system, you only need to reference the system by typing in the variable name, 'system'. This is simply because the function `part_system_create();` Returns an ID of the new system. So 'system' has captured this ID, and now, we have given a sensible name for the system.

All particle systems have their own settings. These settings can be changed through some more `part_system` functions.

```
system = part_system_create();
part_system_automatic_draw(system,true); //Sets whether the particles
will automatically be drawn onto the screen without first calling a
function. If you do not execute this function, a default value of
FALSE is given.
part_system_automatic_update(system,true); //Sets whether the
position/state of particles will be updated automatically. This is
given a default value of FALSE.
part_system_clear(system); //Resets the particle-system. This resets
all the settings to their default values, and destroys all filters
and types placed inside it.
part_system_depth(system,0); //Gives a DEPTH for the particle-system.
Particles need to be given a depth, right? So this value determines
when to draw particles under/over other things.
part_system_destroy(system); //Completely removes the
particle-system from the memory. This includes all
filters and types.
part_system_draw_order(system,true); //Whether to draw
new particles behind old ones, or vice versa. An
example to the left (darker the particles, the older)
shows this.
part_system_drawit(system); //Redraw the particles.
This is only needed if you've set automatic_draw to
FALSE.
part_system_exists(system); //Returns whether a
particle-system with a given ID exists. You'd put in
"system", obviously, because the value of system is the ID of the
system (remember system=part_system_create()?)
part_system_position(system,x,y); //Rarely used, and
you should really forget about this function. Emitter
Filters are used 99% of the time to produce particles
at the right position, NOT systems.
part_system_update(system); //Rarely used when
automatic_updating has been set to true. However, it
may be useful if you want to give particles a "head
start" before they're first drawn. For instance in
this diagram (to right), all faded-circles are
previous positions of the particle, however the non-
```

Old to New (default):

New to Old:



```
faded-circle has been the only DRAWN position of the particle.
```

Obviously, when learning a group of functions, you cannot learn them like ‘that’. They take a bit of practice, referencing the manual over and over until you finally know the function and its arguments off by heart. Well, that’s acceptable; but it takes patience. These functions (and all the rest in this tutorial) will take some time to learn; but will benefit you greatly in the end.

Creating an Emitter

Once you've got your system setup how you think you'd like it (remembering the trial and error factor; you'll probably go back to the system functions many times before you get it how you like it), it's time to create an emitter. Think of emitters like big machines. They're built in different shapes and sizes, and placed somewhere on the room. Then, they shoot out particles. Functions for emitters allow you to choose *how* the particles are emitted. The minimum X/Y coordinates, the maximum X/Y coordinates, and whether the particles are emitted in a *linear* form or *Gaussian* form are just examples of various options you can set.

Now, creating an emitter is very similar to creating a system. You still trap the ID of the new emitter into a variable; however, the creating function for emitters have an *argument*. This argument asks you what system you'd like to place the emitter in.

```
emitter = part_emitter_create(system);
```

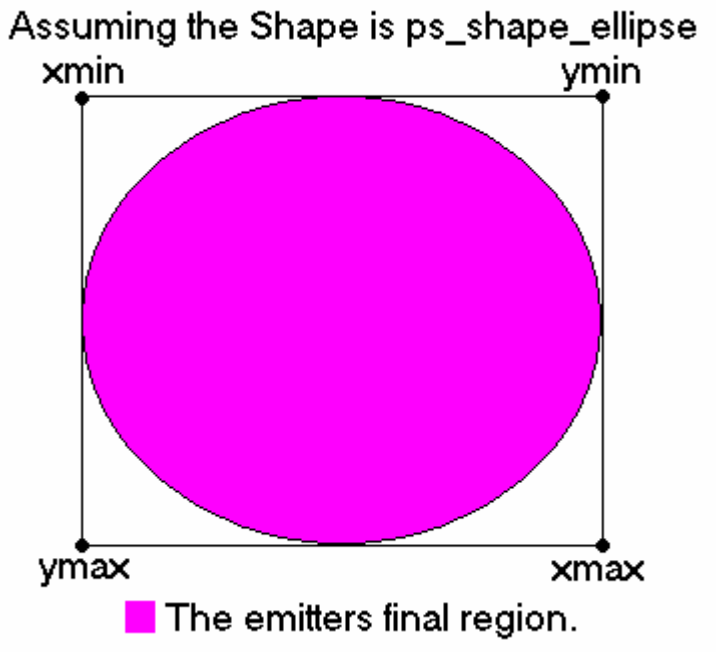
So you've created a new emitter called "emitter", and you've placed it in the system called "system". This emitter can only be used inside this system. If the system clears (`part_system_clear`) or gets destroyed (`part_system_destroy`), the emitter goes with it. You then need to give the emitter some REGION settings. The *region* settings tell the emitter where to create particles, and in what *shape* that area is cut to.

```
part_emitter_region(system,emitter,0,room_width,0,room_height,ps_shape_rectangle,ps_distr_linear);
```

As you can see, the functions are starting to get a little more interesting. Let's first say what this function is doing. It's saying "allow the emitter 'emitter' in the system 'system' to create particles anywhere in the room; distributing every part evenly". Let's look at the arguments to see how that works out.

`Argument0` – The system that holds the emitter you'd like to apply this function to (in this case, 'system')
`Argument1` – The emitter inside that system you'd like to apply the function to (in this case, 'emitter')
`Argument2` – The "minimum X" position that particles being burst by this emitter can be created in. Seeing as the value is 0, no particle can be created from this emitter any further to the left than the left-edge of the room.
`Argument3` – The "maximum X" position that particles being burst by this emitter can be created in. Seeing as the value is `room_width`, we find the width of the room (which is also the right-most pixel in that room), and tell particles that the furthest to the right they can be created is the right-edge of the room.
`Argument4` – The "minimum Y" position. The same as "minimum X" but now we're looking at the Y axis; so the particle can't go any further above the top of the room.
`Argument5` – The "maximum Y" position. Same as "maximum X" but now we're looking at the Y axis; so the particle can't go any further below the bottom of the room.
`Argument6` – ps_shape_rectangle is a constant that equals a number. That number is simply a key to say that the shape of the region is a rectangle. Alternative shape constants are:

```
ps_shape_diamond
ps_shape_ellipse
ps_shape_line
```



To the left is an example of how shapes implement into X/Y mins and max's.

`Argument7` – The final argument determines the chance of particles being created in certain areas of the region. As you probably have noticed by now, particles, when being created inside this emitters region, will be created in a **random position** within that region. So, here are the two options (constants) for the **distribution** argument:

```
ps_distr_linear
ps_distr_gaussian
```

Linear distribution means that everywhere in the region has exactly the same amount of chance those particles will be created in its spot. As for Gaussian, there's more chance that particles are created in the **centre** of the region than there is at the **sides** of the region. Of course, in between the centre and the sides the particles would have a **moderate** chance of being created there.

Although there are many more emitter functions, here are the last two functions that are *required* to have extra attention put to them. These are the functions that create certain amounts of certain particles into the emitters region. Here they are!

```
part_emitter_burst(system,emitter,particle0,5);
part_emitter_stream(system,emitter,particle0,1);
```

The first and second arguments work just the same as most emitter functions. The first argument indexes the system where as the second argument indexes the emitter to create particles in. The third argument is the ID of the particle to create, we'll get into that in later chapters (although it's rather straight forward; just like IDs for emitters and systems), and the final argument determines how many particles there should be to create. So, what's the difference between "burst" and "stream"? Well, burst allows you to "burst" a certain number of particles once, on that step that the function is executed. This is most commonly used, as once you stop executing the function, particles will stop coming out (obviously). Now, stream is the opposite. You only should use the stream function once, and particles will continually leak out of the emitters region, every step. The only way you can stop it is by destroying the particle, assigned emitter, or system. The advantage from using stream over using burst in the step event (Which as you might've guessed, both work slightly the same), is that you can use **negative** values for the number of particles to be burst. How does that work, you ask? Well, it gives a chance of particles being burst. Instead of say bursting 2 particles every step, you could have -2 to have an average of one particle burst every 2 steps.

Now that you've learnt the most important functions for emitters, here are the rejected functions that aren't used as often:

```
part_emitter_clear(system,emitter); //Restores all the emitter
settings to its default settings.
```

```
part_emitter_destroy(system,emitter); //Destroys the emitter from the system.
part_emitter_destroy_all(system); //Destroys all emitters inside the particle system.
part_emitter_exists(system,emitter); //Returns whether the emitter inside the system exists or not.
```

Next, we'll go into the most important step; designing your own particles! After you've learnt that, we'll go into the final 4 filters: destroyers, changers, attractors, and deflectors. These are used only on occasions; sometimes for particle collisions (as they were used in the particle-collision engine contained in the GCE3.X package), or sometimes just for certain effects to be made. They all affect the particles "time in the house". They have regions just like emitters; so I guess you can guess the rest in terms of what each of them do.

DESIGNING YOUR FIRST PARTICLES

So now, it's time to design your first particles. This is the real thing! For each particle type, you give all the information about how the particle moves and looks; and then, those particles are created by the emitter. Any motion changes or physical-changes (for instance turning a particle into another particle with a *changer* filter) are done by the filter. However, the initial information is decided through particle-design functions. Let's create our first particle.

```
particle0 = part_type_create();
```

Just like creating a system, we've created our first particle. As you can see, particles do not belong to certain systems. During the progress of typing this tutorial I accidentally stated that filters *and particles* are part of the system; so I've had to remove that when I came to this part. Why have I told you that? You need to remember that systems only hold filters. Those filters grab certain particles and according to its masters settings (the system) as well as its own settings, affect particles being created and/or changing their path/life while they're moving around the screen.

The next set of functions we're going to look at is the customization functions. These functions input certain pieces of information about the particle type. That includes things like direction, speed, colour, shape, angle, and much more. We'll need to look at each function in detail, not just give comments about them like previous groups of functions for *filters* and *systems.*

ALPHA FUNCTIONS

```
part_type_alpha1(particle0,alpha1);
part_type_alpha2(particle0,alpha1,alpha2);
part_type_alpha3(particle0,alpha1,alpha2,alpha3);
```

These are the functions that deal with the alpha/transparency value of the particle. As in all particle functions, the first argument asks for the ID of the particle type you wish to input information into. Again, as you know, the value of the variable "particle0" is the ID that we need. Now, let's look at these functions. The first function, `alpha1`, sets the particle to a permanent alpha value. This alpha value is carried out all the way from the particles birth (being created by an emitter), during it's life (moving around the screen), and it's death (changed or destroyed by a filter, or it's life value has run out). The second argument asks for this permanent alpha value. For those who have not dealt with alpha values, 0 alpha makes the particle completely invisible, 1 will make the particle completely visible, and any fractions in between are used appropriately (I.E 0.5 makes the particle semi-visible).

The second function, `alpha2`, sets the particle to 2 alpha values. The second argument asks for the first alpha value. This is the alpha value that the particle will have when it is born (created). During its life, it will slowly fade to the second alpha value (`argument2`/the third argument), and will have died just after the alpha value has reached the second alpha value.

The final alpha function, `alpha3`, is pretty obvious as you see from the `alpha2` function. Quite basically, `alpha3` works just the same as `alpha2`, however the alpha first fades from the beginning value to the middle value (`argument2`), reaching that middle value in the prime of the particles life (as in, right the middle), and *then* fades towards the final value (`argument3`).

Blending and Colour Functions

The next group of functions deal with the colour(s) of the particle, and the additive blending of the particle.

```
part_type_blend(particle0,additive);
part_type_color_hsv(particle0,hmin,hmax,smin,smax,vmin,vmax);
part_type_color_rgb(particle0,rmin,rmax,gmin,gmax,bmin,bmax);
part_type_color_mix(particle0,color1,color2);
part_type_color1(particle0,color1);
part_type_color2(particle0,color1,color2);
part_type_color3(particle0,color1,color2,color3);
```

These functions get a little more complicated. The first function is rather simple. The second argument is a true/false value, asking whether to give additive-blending to the particle or not. Additive blending adds the colours behind the particle to the particles colour itself. It gets rather hard to explain, however it's good to play around with, especially with explosions.

The second function (`color_hsv`) sets all particles to one fixed colour, however that colour ranges from particle to particle (as in, each particle will have a different colour). These colours are ranged from a minimum hue value to a maximum hue value, a minimum saturation value to a maximum saturation value, and a minimum luminosity value to a maximum luminosity value (HSV).

The third function (`color_rgb`) works very similarly to the second, however the min/max ranges are determined from red/green/blue (rgb) values, instead of hue/saturation/luminosity (hsv) values.

The fourth function, `color_mix`, works slightly the same as the other colour functions we've covered, in the fact the particles have a fixed colour for their whole life, and that it's randomized. However, the colours aren't determined via ranges, but simply 2 solid colours to choose from. For instance, instead of using rgb/hsv functions to allow particles to have any shade from red-to-dark-red, we can allow particles **only** to have either red or dark red.

`Color1`, `color2`, and `color3` works very similar to the `alpha1/2/3` functions, however this deals with colour. `Color1` gives every particle one colour throughout its life. `Color2` allows the particle to start off at one colour, and end with another, fading from one colour to the next during its life. And `color3` gives the particle 3 colours, fading from each one throughout its life.

Particle-Shaping Functions

This can be one of the most creative set of functions to use when creating particles. They deal with the primary shape/sprite of the particle. That includes size/angle.

```
part_type_shape(particle0,shape);
part_type_sprite(particle0,sprite,animate,stretch,random);
part_type_orientation(particle0,ang_min,ang_max,ang_incr,ang_wiggle,a
ng_relative);
part_type_size(particle0,size_min,size_max,size_incr,size_wiggle);
part_type_scale(particle0,xscale,yscale);
```

The first function in this subchapter is the most used. You'll almost never make a particle without needing it, unless you want the particles to be a plain old pixel flying around the screen. This pretty much sets the particle to one of a selection of shapes pre-built into Game Maker. These shapes usually work for any type of particle effect

you want to create, however, going more advanced will mean having to create your own particle shape and using the `part_type_sprite` function, but we'll get into that. The shapes are determined through a list of constants. You simply put one of these constants into `argument2` of `part_type_shape:`

```
pt_shape_circle
pt_shape_cloud
pt_shape_disk
pt_shape_explosion
pt_shape_flare
pt_shape_line
pt_shape_pixel
pt_shape_ring
pt_shape_smoke
pt_shape_spark
pt_shape_sphere
```

No use me explaining what each shape is like. Even if you don't understand it by the constant name, it just takes a game-test to see what it looks like (remember what I said about trial and error?).

The sprite function takes a bit of graphic-design knowledge, however is really powerful for making your particle effects look *much* nicer, assuming the pre-made shapes aren't good enough for your purpose. The second argument asks for a sprite-ID/name. You then determine, from the other arguments, whether to animate the sprite with its sub images, whether to stretch that animation over the lifetime of the particle (so it only plays once, finishing at the particles death), and whether to start the animation (or the freeze frame) from a random subimage. Remember, if you want some nice alpha-mapping for the sprite you use here, it's good to use the `sprite_set_alpha_from_sprite` function. This is very useful for `part_type_sprite`, however I'm not going to teach you how the function works.

The next function deals with the angle (like `image_angle`) of the particle. There are lots of nice little options here, let's go through each of them.

`ang_min` – The minimum angle (in degrees) for the particle
`ang_max` – The maximum angle (in degrees) for the particle (so obviously it ranges, randomly)
`ang_incr` – You can make the angle increase by a certain amount every step (and use negative values)
`ang_wiggle` – "Wiggling" a value simply makes it increase the amount you specify, then move backwards. Kind of like a pendulum motion.
`ang_relative` – A good function, allowing you to have the angle point relatively to the direction/path of the particle (it will also be relative to the arguments above, guess that's a matter of trial/error)

The next function is also important, and as you can see from the arguments works similarly to the orientation function, however this deals with particle size, not angle. You specify the minimum/maximum size values (initial values), 1 being full-size, and then you choose the increment of the size per step, and of course, size "wiggle". No size-relative argument here, unlike the orientation, but you would assume that makes sense.

The final function is a bit like the size, however this is with both xscale *and* yscale, not just both. This is used as a factor to the normal size, and is rarely needed. However, as you can easily see, it takes 2 arguments to determine both xscale and yscale.

Particles Life/Death Functions

The next group of functions deal with how long the particles live, and child-particles they make.

```
part_type_life(particle0,life_min,life_max);
part_type_step(particle0,step_number,step_type);
part_type_death(particle0,death_number,death_type);
```

The first function is a rather important function, assuming the default life values aren't enough for you. This function gives a minimum and maximum life-value for the particles. Each particle, when born, will be given a random life value, ranging from the `life_min` argument and the `life_max` value. Of course, just like humans, particles can die from accidents too (like colliding with filters that change them or destroy them).

The second function, `part_type_step`, allows particles to make children in each step of its life. Pretty much, you can choose how many children to make in each step with the `step_number` argument (or like with the `part_emitter_stream` function, use negative values to give a chance of children being created in each step), and you can choose what particle-type that child is, with the `step_type` argument. This is good for say trail effects on rockets, etc.

The final function (`part_type_death`), allows particles to give a certain number of children when it dies. Good for say smaller particles (like a water-splash). This works the same as `part_type_step` arguments-wise, except the negative-values randomly decide whether to give children at all when dying.

Speed and Motion Functions

The final group of functions deal with the motion of the particles. Including direction, gravity, and speed.

```
part_type_speed(ind,speed_min,speed_max,speed_incr,speed_wiggle);
part_type_direction(ind,dir_min,dir_max,dir_incr,dir_wiggle);
part_type_gravity(ind,grav_amount,grav_dir);
```

The first two functions work very similarly to `part_type_size`. You can simply choose a minimum/maximum speed (or direction), an increment-per-step for speed or direction, and a wiggle amount for speed/direction.

The final function we're going to deal with is `part_type_gravity`. This changes the particles direction with force according to the `grav_amount` (the force of the gravity), and `grav_dir` (the direction, in degrees, the force pulls towards).

Other Functions

The final functions for `particle_type` functions deal with clearing, destroying, and some conditional functions.

```
part_type_clear(particle0);
part_type_destroy(particle0);
part_type_exists(particle0);
```

The first function clears all the particles settings to its default. The second destroys the particle altogether, good for saving memory. The final returns `true` or `false` according to whether the indicated particle type exists.

Adding Optional Filters

Besides emitters, there are a few filters that are great to use for different effects, particle-collisions, and to simply restrict particles from certain areas. These optional filters are pretty easy to use, so they don't need such a large chapter for them.

Attractor Filters

Attractor filters are forces, slightly like magnets, that can either pull particles towards them, or push them away. These are good for particle collisions, like bouncing or something alike. The manual strictly states to use few of these, as they slow down processing of particles. Creating an attractor is very similar to creating an emitter:

```
attractor = part_attractor_create(system);
```

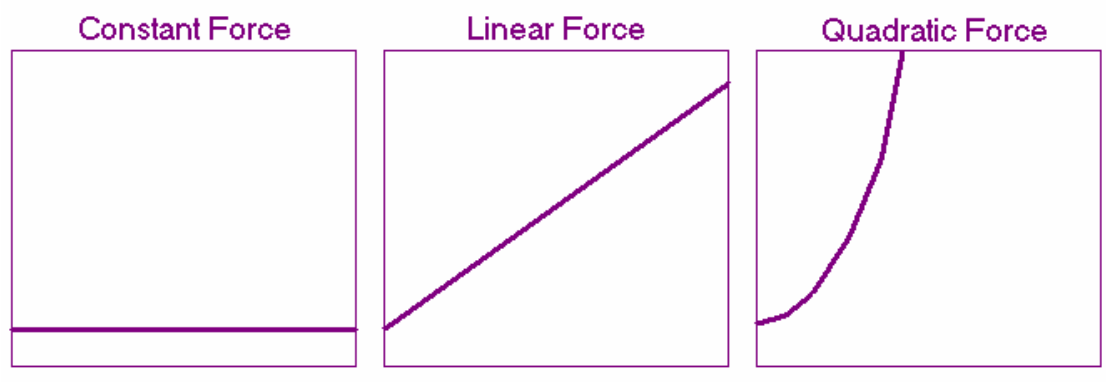
With this newly created attractor, you can now use these two functions to customize your attractor:

```
part_attractor_force(system,attractor,force,dist,kind,additive);
part_attractor_position(system,attractor,x,y);
```

The first function sets the type of force the attractor has on particles. Like the emitter (and all other filters), the first argument asks for the system the attractor is located in, and the second argument then asks for the attractor ID itself. Then, we go into the force argument. The `force` is a real; the larger the value, the faster the particles accelerate towards the attractor. You can of course use negative values to make particles fly away from the attractor. We then go into the fourth argument, which determines the maximum distance that particles need to be to the attractor before they start being attracted towards the attractor (or detracted). Then we have the `kind` of force the attractor has. A list of constants are available for this argument:

```
ps_force_constant
ps_force_linear
ps_force_quadratic
```

This is the *way* that the attractor attracts particles. Constant force attracts/detracts particles at the same rate from its maximal distance to its minimal distance. Linear force makes the force value smaller as it goes further away from the minimal distance (minimal distance being the exact position of the attractor). This means that when a particle first enters the maximal distance, it will be attracted a lot less than say the middle of the distance or at the attractors' position. **Note:** The force-value you give in this function will be the force applied to particles when they are at the minimal distance. Finally, we have the quadratic force. This force is rather mathematical to describe, but basically, the force does not grow at a steady rate like linear; instead the increment that the force grows by also increases. Here's a diagram of the three forces:



The final argument focuses on how the particle takes that force. This is whether it is `additive` or not. If the force is additive, the particle's direction and speed is affected by the force. This means if it is travelling away-ish from the attractor, and gets caught into the force, it will slow down, then speed up the other way, as the direction slowly changes. However, if `additive` is set to `false`, the particles position is all that's affected. This means he will not take any fight to get away from the force, but instead snap to the correct direction/force-speed that the attractor asks for.

The second function, `part_attractor_position` is rather simple. The `x` and `y` arguments determine the position of the attractor. This is also the "minimal distance" for the force.

Quite simply, you know have a nice working attractor. You can also use these very standard functions for filters, to destroy attractors, take conditions on them, etc.

```
part_attractor_clear(system,attractor);
part_attractor_destroy(system,attractor);
part_attractor_destroy_all(system);
part_attractor_exists(system,attractor);
```

The first function clears all the attractors' settings to its default. The second destroys the indexed attractor. The third destroys all attractors in the indicated system, and the final returns `true` or `false` according to whether the indicated attractor exists.

CHANGER FILTERS

Changer filters allow you to change one type of particle into another one while it's alive. These can be good for various effects, for instance filtering bigger particles into smaller particles when falling through floors, etc.

```
changer = part_changer_create(system);
```

This creates a changer into the system. We now look at the functions that will input settings into this changer.

```
part_changer_kind(system,changer,kind);
part_changer_region(system,changer,xmin,xmax,ymin,ymax,shape);
part_changer_types(system,ind,parttype1,parttype2);
```

The first function allows you to choose how much of the particle is changed. This is determined by one of three constants:

```
ps_change_motion
ps_change_shape
ps_change_all
```

The first constant, `_motion`, will change only the motion-related settings from one particle to another. This means the particle will stay the same appearance, but its speed, direction, gravity, etc will be changed to the other particle's parameters. The second, `_shape`, is the opposite. All motion parameters are kept the same, but the look of the particle (shape, colour, size, etc) will be changed to the other particle-types parameters. The final one switches all parameters over to the other particle.

The next function focused, quite frankly, on the region of the changer. This works just like the emitter, except there are no distribution constants (for obvious reasons). The shape, just like the emitter, determines from the `ps_shape_` constants.

The final function asks you to choose what particle-types to change (when they collide with the changers region), and then what particle-type to change it to. This is decided from `parttype1` and `parttype2`. This function is obviously rather important.

The last functions we need to deal with are the functions you see in all other filters…

```
part_changer_clear(system,changer);
part_changer_destroy(system,changer);
part_changer_destroy_all(system);
part_changer_exists(system,changer);
```

First clears all the settings from the changer, the next destroys the changer, the third destroys all changers from the system, and the final returns whether the specified changer exists or not.

DEFLECTOR FILTERS

Deflector filters are very useful, especially for particle colliding. They, quite basically, deflect any particles coming into the region. The way the particles deflect on the deflectors' `kind` and `friction` and of course the particles motion towards the region. Deflectors are nice and fast, unlike attractors, so you can place lots of them without too much CPU being used.

```
deflector = part_deflector_create(system);
```

This, as always, creates the deflector into the specified system. After this, we need to input settings into the deflector.

```
part_deflector_friction(system,deflector,amount);
part_deflector_kind(system,deflector,kind);
part_deflector_region(system,deflector,xmin,xmax,ymin,ymax);
```

The first function allows you to choose a friction amount for the deflector. This requires a bit of trial & error to get it working, but it's rather straight forward. The second function asks you for the kind of deflector. This is answered with one of two constants.

```
ps_deflect_horizontal
ps_deflect_vertical
```

The first deflects particles horizontally (used for horizontal walls mostly), and the second deflects particles vertically. I'm not 100% sure why there's not a `ps_deflect_both`, but I would assume there's a legit reason for it, I've just not thought about it much. Anyway, the final function, again, is the region for the deflector. The shape is always rectangle now, unfortunate, but it's the rules. So in this case there is no `shape` argument.

```
part_deflector_clear(system,deflector);
part_deflector_destroy(system,deflector);
part_deflector_destroy_all(system,deflector);
part_deflector_exists(system,deflector);
```

The first function will clear all the settings from the deflector (to its default settings), the second will destroy the deflector, the third will destroy all deflectors from the system, and the final function returns whether the specified deflector in the specified system exists or not.

DESTROYER FILTERS

The final filter we'll look at is a simple one. Destroyers destroy and particles that come into its region.

```
destroyer = part_destroyer_create(system);
```

This creates a new destroyer into the system. The simple thing about destroyers is that the only settings we need to give, is the region.

```
part_destroyer_region(system,destroyer,xmin,xmax,ymin,ymax,shape);
```

And I'm sure by now you know how this works. Again, use the emitters' constants for the shape.

```
part_destroyer_clear(system,destroyer);
part_destroyer_destroy(system,destroyer);
part_destroyer_destroy_all(system);
part_destroyer_exists(system,destroyer);
```

Again, I'm sure you know what these do. No use explaining; Besides, I guess you need a bit of a challenge to remember them. Helps you learn.

FINAL REVISION

For the past 15 pages, assuming you haven't fallen asleep at your desk, has focused on helping you design your own particles. We've gone through systems, filters, and particles themselves. You need to remember that learning a new part of any part of the manual for GML can take a bit of practice. Practice will build you a logical structure of how particles work, and then it's just a matter of trial and error.

This tutorial was written as a means for you to read from top to bottom. For quick reference for functions/arguments, it's best that you read from the GML Manual (F1 during Game Maker's runtime) instead of revising this whole tutorial. However, if you feel you haven't got the hang of things, feel free to read it over anyway.

Particles are wonderful things, but they also have their limits and you need to learn what those limits are, when those limits can be secretly exceeded, and when it's good to leave particles out completely. Never-the-less, you can be rest-assured that you've spent a good 5 hours on this damn thing.

Enjoy
Rhys Andrews
GameCave Productions
http://www.gamecave.org