CBEffects for Corona® SDK

The Ultimate Free Particle Engine

Usage Manual

Part 1 | General Information

Miscellaneous

If I write something in red text, that means otherwise an error will occur. If the text is **bold and red**, that means that otherwise the system will crash.

Credits, samples, and the like are in the CBResources folder for CBEffects.

What You Get

The CBEffects package is a folder containing everything that is needed - or unneeded - for CBEffects to function. It has two ".lua" files, a basic texture set, a CBInfo file that gives information such as version history, bug fixes, etc., and a CBReference file that gives parameter explanation. The whole caboodle is a little over 300 Kb in size.

Why You Should Get It

CBEffects is a particle engine library that is extremely flexible, easy to use, and generates moderate to low memory overhead. It takes a completely different slant to make particle engines completely customizable.

For example, instead of limiting users for particle types - a set number of possible parameters like width, height, etc., and then creating particles itself, it lets the user simply specify **how to create the particle, period.** This is done with a parameter called the "build". The "build" parameter is a function that returns a display object - put simply, **anything you can make in a function can be a particle.** Limited by nothing. So if you are trying to do something that is difficult with set-parameter particle systems - something like creating an animated sprite with a width that varies between 100 pixels and 500 pixels and then after the particle engine has emitted particles 50 times change the height to a constant 200 pixels and the spritesheet file to a different one and the reference point to topLeft and then the play speed for the sprite to 10 milliseconds between plays - you can do that easily with CBEffects. Because the build function is just that - a function - you can do **anything** with it. For example:

```
build=function()
  local width=100
  local height=math.random(100, 500)
  return display.newImageRect("myImage.png", width, height)
end

build=function()
  return display.newRect(0, 0, 100, 100)
end
```

```
build=function()
  local shape=display.newRect(0, 0, 80, 80)
  shape.strokeWidth=math.random(4, 9)
  return shape
end
```

So in the end, particles can be anything you can possibly imagine.

And that's not the only way that CBEffects is different than most particle systems. There are many more values that can also be created with functions - things like particle color, positioning of the particle on creation, particle velocity, etc.

CBEffects is also free of charge to get, modify, or do whatever you want to do with it. You may **under no circumstances** sell it for any amount of money or anything else that you might want to sell it for, period.

It comes with a library of presets, which are tables containing data to make vents you might want to use, like snow, rain, fire, etc. In other words, if you want a "fire" effect, you don't have to make it yourself. You can load the fire preset and change whatever values you want, but it will start with values that make a fire effect. To load a preset, specify the "preset" parameter in the VentGroup function. (see "Presets")

Why You Shouldn't Get It

CBEffects is a folder, not a single ".lua" file. It is also a bit larger in size than some particle systems (about 340 KB).

Moreover, CBEffects is **not** nest-able. You cannot put the CBEffects folder inside of another one.

Last but not least, I **have only commented a tiny bit of the code**. So if you want to make your own addition, you're going to have to figure out how it works yourself.

I apologize if this section sounds biased, but, then, I did make it :)

Particle Effects With CBEffects

With CBEffects, effects are created with **CBObjects**. A CBObject can be either a **CBVentGroup** or a **CBFieldGroup**. CBVentGroups contain **vents**. A vent emits a particles with options you made earlier, like gravity, color, etc.

A CBFieldGroup contains **fields**. Fields modify particles emitted from vents. With fields, you can do things like bounce particles off of walls, change their color when they enter different zones, etc. They operate by doing a function when a particle collides with them.

Vents can exist and work as expected without fields, but fields cannot work without vents.

Localization

CBEffects takes a different approach to particle engines - localization of VentGroups and FieldGroups. Instead of keeping track of all of the engines internally, CBEffects works by giving you a handle to the VentGroup or FieldGroup, and you start, stop, or modify it directly **with the handle**, instead of **through the library**.

Getting Set Up

Setting up CBEffects is as simple as dragging the CBEffects folder into your main project folder **-** no nesting **-** and requiring the library.

```
local CBE=require("CBEffects.Library")
```

The file that does all of the converting from you calling the function to the effect on screen is the “Library.lua” file. The other file is a helper library, and will generally not be of interest to you.

Now that CBEffects is included and loaded, you can start making particle engines!

Part 2 | CBVentGroups

Making CBVentGroups

CBVentGroups are CBEffects' name for particle engines. They emit a specified particle type with various parameters. They are made with one method - the VentGroup function.

```
local VentGroup=CBE.VentGroup{
  <values>
}
```

Of course, the "<values>" is just a placeholder for what would really be in it.
Inside the VentGroup function's **master table** (the main table holding all of the data) are the options for creating vents. Each vent is specified with a separate table:

```
local VentGroup=CBE.VentGroup{
  {},
  {}
}
```

In that example, there will be two vents inside the VentGroup. Blank vent parameter tables automatically load the default preset, which brings me to

Presets

CBEffects comes with a ParticleHelper class containing preset vent data. They include every parameter that a VentGroup will ever need. This makes things easier for you - instead of starting from scratch each time you need a new effect, you can load the preset and change whatever values you need to, but it starts out with all of the values already. To use a preset, specify the "preset" parameter in the parameter table for a vent:

```
local VentGroup=CBE.VentGroup{
  {
   preset="burn"
  }
}
```

When you specify the "preset" parameter, it's basically the same thing as going to the ParticlePresets file, copying a preset, and putting it inside of the parameter table. However, instead of taking up so much space and having to delete and rewrite each parameter if you want to change it, CBEffects keeps track of the preset you have and uses that. It actually uses something that goes a bit like this (I'll just use the "build" parameter as my example):

1. Check for the parameter "build" in the user's parameter table
2. If it's there, use it as the "build" parameter
3. If it's not there, use the preset's "build" parameter.

Presets can save you a lot of time. With the presets, you will **never** need to specify every parameter. That's quite helpful, knowing that there are 55+ possible parameters.

Parameter and Value Explanation

Here I'll go in-depth into parameters that deserve more space than 5 lines in the parameter documentation and values that may be a little confusing or difficult to see the effect of.

The Alpha Parameters
(parameters "alpha", "endAlpha", and "startAlpha")

The alpha parameters are measurements of the alpha each particle is at different points on it's timeline. When the particle is created, it has "startAlpha" as it's alpha. It instantly transitions to "alpha" in the "fadeInTime", and once it's finished with the "fadeInTime" transition, transitions to endAlpha, taking "lifeStart" delay and "lifeSpan" time to finish.

Visual:

```
startAlpha      -->       alpha      <delay>       -->       endAlpha
             fadeInTime              lifeStart   lifeSpan
```

The onCreation, onUpdate, and onDeath Parameters
(parameters "onCreation", "onUpdate", and "onDeath")

The onCreation, onUpdate, and onDeath parameters are one of the things that makes CBEffects so customizable. They are functions that each execute at a different time - the onCreation function executes when the particle is created, the onUpdate function is executed on EnterFrame, each time the ParticlePhysics library updates the position of the particles, and the onDeath function executes when the particle is removed. For each function, the particle, the particle's parent vent, and the particle's content group is passed.

```
onCreation=function(particle, vent, content)
  particle.strokeWidth=math.random(1, 8)
  vent.x, vent.y=math.random(500), math.random(500)
  content.rotation=content.rotation+1
end

--Example for onUpdate:
onUpdate=function(particle, vent, content)
  particle.strokeWidth=particle.strokeWidth+0.5
  print(particle.x, particle.y, particle.width, particle.height)
end
```

The above example could be used for debugging, for example - print stats about the particle, the vent, the content, or anything that belongs to it. Not only could you use it for debugging - if you can print stats about it, you can change it:

```
onUpdate=function(particle, vent, content)
  particle.thisValueThatYouCreatedSomewhereEarlier=math.random(100,10000)
end
```

And, finally, the onDeath function. It is important to note that the onDeath function executes right before the particle is destroyed, and “right before” meaning that the particle will be destroyed instantly after that. So no transitioning or timers for the onDeath function.

The Content Group
(vent value)

The content group of a vent is a group that holds all of the particles. It can be used to represent the particles as a whole - you can do something to it and it will affect all of the particles. For example, if you set the content group’s alpha to 0.5, everything in it will never have an alpha over 0.5. It’s just a group. So anything that usually happens with groups will happen with the content group. Moving the content group will move every particle along with it, rotating it will rotate the particles according to the content group’s reference point, etc. It is simply for referencing the vent’s particles as a whole.

The PropertyTable
(parameter “propertyTable”)

The PropertyTable is a table of values for each particle. When the particle is created, it iterates through the PropertyTable and assigns each value to the particle. For example, if you want to add values like “blendMode”, “strokeWidth”, or custom values you want to refer to later, you can use the PropertyTable by specifying all of those:

```
propertyTable={
 blendMode=”add”,
 strokeWidth=4,
 aCustomValueIWantToReferToLater=”CBEffects is Awesome!”
}
```

For varying values, you should specify them in the onCreation function:

```
onCreation=function(particle, vent, content)
 particle.varyingValue=math.random(1,500)
end
```

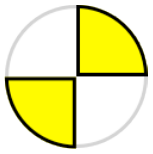
The Angle Parameters
(parameters “angles”, “preCalculate”, “autoAngle”)

The angle parameters specify what direction the particles will travel.

The “autoAngle” parameter is a boolean value that sets whether the angles table contains numbers or tables. If it is set to “true”, then the angles table must hold tables containing the low angle and the high angle for however many angle sets you want. For example:

```
autoAngle=true,
angles={
 {0, 90},
 {180, 270}
}
```

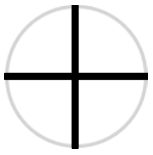
Will result in this (angles 0-90, 180-270):



While this:

```
autoAngle=false,
angles={
 0, 90, 180, 270
}
```

Will result in this (angles 0, 90, 180, 270):



When using angle tables (autoAngle=true), the first value must be smaller than the second value.

PreCalculate is a boolean parameter which determines whether angle velocities are calculated at the start or are calculated each time the particle is created. In other words, if you want to change the angles particles will travel on or the velocity of the particles, you should set it to “false”, otherwise set it to “true”. Setting it to “false” generates more memory.

Note: Not pre-calculating angles means you can change them after creating the vent, however, changing them does not consist of changing the “angles” parameter. Change the “velAngles” for the angles to change.

```
--preCalculate=false
MyVentGroup:get(“MyVent”).velAngles={90, 180, 270, 360}
```

The PositionType Parameter
(parameter “positionType”)

This parameter specifies how the particles will be positioned when they are created. There are a number of strings you can use for it, or you can use a function.

“inRadius”
Setting it to “inRadius” means that particles will appear inside of a specified radius. You can customize this radius with the parameters “posRadius” and “posInner”. The PosRadius is how big the radius will be that the particles appear in; the PosInner is how big the radius will be that the particles appear outside of. Setting the PosInner to 1 makes the particles appear inside of a circle, setting it to something higher makes them appear inside of a ring. **Do not set the PosInner to be higher than the PosRadius.**

“atPoint”
“atPoint” means that particles will appear perfectly at the vent’s X and Y position on-screen.

“alongLine”
Setting “alongLine” as the positionType will make particles appear along a line drawn from parameter “point1” to parameter “point2”. Point1 and point2 are tables with X, Y values in them:

```
positionType=”alongLine”,
point1={0, 0},
point2={1024, 768}
```

"fromPointList"
Chooses from a table of points containing {x, y} values:

```
positionType="fromPointList",
pointList={{0,0}, {100,100}, {200,100}}
```

With fromPointList, you can also specify the "iteratePoint" and "curPoint" parameters. The iteratePoint parameter is boolean, and specifies whether to choose randomly from the pointList or to go one by one through it. The curPoint parameter specifies what point to start with.

"inRect"
The inRect positionType positions particles inside of a rectangle. The dimensions of the rectangle are defined with rectLeft, rectTop, rectWidth, and rectHeight.

Using a Function as the PositionType
Simply use a function that returns two values - X and Y. The particle, the particle's parent vent, and the vent's content group are all function arguments.

```
positionType=function(particle, vent, content)
  return vent.x-content.x, vent.y-content.y
end
positionType=function()
  return math.random(500), math.random(500)
end
```

The Physics Table
(parameter "physics")

The physics table contains all of the physics data for the vent. Every parameter mentioned in this section will be inside the physics table, unless the parameter is in **bold face.**
Important: All of the angle parameters are inside of the physics table. (see "The Angle Parameters")

"velocity"
How fast the particle will travel in the specified angles (see "The Angle Parameters")

"velFunction"
A function that returns X and Y velocity for a particle. You should use this if you want more than angle control over the particle's velocity. To use it, you need to specify the parameter "useFunction" as the boolean value "true". The particle,

the particle's parent vent, and the vent's content group are all function arguments.

```
velFunction=function(particle)
  if particle.x<display.contentCenterX then
    return -5, 0
  else
    return 5, 0
  end
end
```

"divisionDamping"
Boolean, specifies whether the damping equation is velocity/damping or velocity-damping.

ParticlePhysics
(a ParticleHelper class)

CBEffects moves particles around on-screen with a class that makes them look "physics-y". This means that gravity and forces can act on particles, instead of having to succumb to the restrictions that happen when using transitions. I did not use real physics equations, I just made equations that looked good on-screen.

All velocities are given in pixels per frame. That means the amount of pixels the object will move every 1/30 of a second, if your FPS is 30, or 1/60 of a second, if your FPS is 60. The presets in CBEffects' preset library are formatted for 60 FPS - if they seem slow or out of place, you probably have your project set to 30 FPS.

You can take the ParticlePhysics class out and use it for something else - in fact, I do that a lot, if I'm not in the mood to hard-code movement for something.

ParticlePhysics has an API that works much the same as Corona's built-in Box2D physics - even if it doesn't do the same things. There are some differences, though:

object:setLinearVelocity(xVel, yVel) - Sets the object's X and Y velocity

object:applyForce(xVel, yVel) - Does not do it over time, like Box2D - adds "xVel" to the object's X velocity and "yVel" to the object's Y velocity.

object:applyTorque(t) - Adds "t" to the object's angular velocity.

object:getLinearVelocity() - Returns object.velX and object.velY. You can also just use object.velX and object.velY if you want.

CBVentGroup Methods

Starting a vent inside of a CBVentGroup is quite simple - first, **make sure you specify a title for each vent you create! This is important!** Once you've created your vents, start them with one of two commands:

```
VentGroup:start("MyVent")
--VentGroup:start("MyVent", "MyVent2", "MyVent3")

--OR--

VentGroup:startMaster()
```

The top example is the generic "start" command. It will start the vent with the title(s) you specified as the function argument(s).

The "startMaster" command iterates through each vent in the CBVentGroup. If you have specified isActive on that vent as the boolean value "true", it will start it. Otherwise, it will skip that vent and continue to the next.

Personally, I use "start" more often than "startMaster". I like being able to start only the ones I specify without going to each and setting isActive to true.

The "start" command can also have as many vent titles as you want passed to it and it will start all of them:

```
VentGroup:start("MyVent", "MyVent2", "MyVent3")
```

Stopping a vent is just as easy as starting one - the same commands apply to them, except this time you use "stop" and "stopMaster". Furthermore, the isActive parameter doesn't affect anything with the "stopMaster" command. Just for clarification, I'll put a stopping example here, too:

```
VentGroup:stop("MyVent")
--VentGroup:stop("MyVent", "MyVent2", "MyVent3")

--OR--

VentGroup:stopMaster()
```

In addition to starting and stopping vents, there are also a number of other commands.

The "emit" and "emitMaster" commands emit one round of particles from vents in a CBVentGroup.

```
VentGroup:emit("MyVent")
--VentGroup:emit("MyVent", "MyVent2", "MyVent3")
```

```
--OR--

VentGroup:emitMaster()
```

You can also emit one round of particles by using the vent’s title:

```
--Vent title is “smoke”
VentGroup.smoke()
```

The “clean” and “cleanMaster” commands destroy all particles from a vent abruptly. Each particle is destroyed, but the **vent itself is not changed in any other way.**

```
VentGroup:clean(“MyVent”)
--VentGroup:clean(“MyVent”, “MyVent2”, “MyVent3”)

--OR--

VentGroup:cleanMaster()
```

You can destroy a vent completely - clean the particles from it and nil out the vent itself - with the “destroy” command. It does the same thing as “clean” but it also removes the vent.

```
VentGroup:destroy(“MyVent”)
--VentGroup:destroy(“MyVent”, “MyVent2”, “MyVent3”)
```

Destroying a single vent should be used when you still want to keep a CBVentGroup but don’t want a particular vent inside of it. For destroying the entire CBVentGroup, use the “destroyMaster” command. It destroys the entire CBVentGroup, not just a single vent. To correctly nil out a CBVentGroup, do this:

```
VentGroup:destroyMaster()
VentGroup=nil
```

If you want to change a value of a vent easily, use the “get” command and change it from there. Unlike most commands, there is no “getMaster” command.

```
local returnedVent=VentGroup:get(“MyVent”)
returnedVent.color={{255, 255, 0}}

--OR--

VentGroup:get(“MyVent”).color={{255,255,0}}
```

You can also pass as many vent titles as you want:

```
local returnedVent1, returnedVent2, returnedVent3=VentGroup:get(“MyVent”,
“MyVent2”, “MyVent3”)
```

And, last but not least, there is the “translate” command. This is more of a convenience function, as it simply saves some typing. All it does is move a vent.

```
VentGroup:translate(“MyVent”, display.contentCenterX, display.contentCenterY)
```

It’s basically the equivalent of this:

```
local v=VentGroup:get(“MyVent”)
v.x, v.y=display.contentCenterX, display.contentCenterY
```

Not to mention it doesn’t create a whole new variable to move.

Vent Methods

Here’s a little section for the few vent methods. These are done to vents once you’ve collected them with the :get() command.

The “emit” command is the base particle function - calling it will emit one round of particles from the vent.

```
MyVent.emit()
```

The “set” command sets all values from a table to the vent. It’s also just a convenience function:

```
MyVent.set{
 color={{0,0,0}},
 positionType=”inRadius”,
 posRadius=100,
 posInner=1
}
```

The “resetPoints” function resets the vent’s table of points if the positionType is “alongLine”. Specify the vent’s point1 and point2 with setting of it or the “set” function, call resetPoints, and the vent’s line is updated.

```
MyVent.point1={0, 0}
MyVent.point2={100, 100}
MyVent.resetPoints()
```

That’s all of the methods for vents, concluding the documentation for VentGroups.

Part 3 | CBFieldGroups

Making CBFieldGroups

If you know how to create CBVentGroups, you'll know how to create CBFieldGroups. CBFieldGroups are created with the "FieldGroup" function.

```
local FieldGroup=CBE.FieldGroup{
  {}
}
```

Does that look familiar? FieldGroups are created exactly the same as VentGroups, including the setting of parameters and quite a lot of the same methods.

Using CBFieldGroups

FieldGroups modify particles from VentGroups. Each FieldGroup has a unique target vent whose particles are susceptible to the FieldGroup. The target vent is not optional. It is most easily specified with the "get" command for VentGroups:

```
local FieldGroups=CBE.FieldGroup{
  {
    title="MyField",
    targetVent=VentGroup:get("MyVent")
  }
}
```

Now the particles from "MyVent" in the VentGroup will be modified by "MyField" in the FieldGroup. Modifying particles is done with a function - the "onCollision" function.

```
--In a data table for a field
onCollision=function(particle, field)
  particle:setFillColor(255, 255, 0)
end

onCollision=function(particle, field)
  particle:applyForce(field.x-particle.x, field.y-particle.y)
end
```

If you're starting timers, transitions, or the like, it's best to use the singleEffect parameter. It only executes the onCollision function once per particle.

```
--Use it for starting transitions, timers, etc.
singleEffect=true,
onCollision=function(particle, field)
  particle.t=transition.to(particle, {velX=500, time=30})
end
```

FieldGroup Shapes

When using FieldGroups, you can specify three collision shapes - rectangular, polygonal, or circular.

Rectangular

Rectangular shapes are specified with the shape value “rect”.

```
shape=”rect”
```

The dimensions of the rectangle are self explanatory from the names:

```
rectLeft=0,
rectTop=0,
rectWidth=display.contentWidth,
rectHeight=display.contentHeight
```

Polygonal

Polygonal shapes are specified with the shape value “polygon”.

```
shape=”polygon”
```

And the dimensions of the polygon are pairs of numbers in the “points” table:

```
points={0,0,  100,100,  90,20,  480,370}
```

Circular

Circular shapes are specified with the shape value “circle”.

```
shape=”circle”
```

And there’s only one parameter for the circular shape: radius.

```
radius=150
```

That’s all of the shapes for FieldGroups.

FieldGroup Methods

FieldGroups have much the same methods as VentGroups - I’ll still list them, I’ll just not explain them. Note that FieldGroups **must be started** for them to modify particles. I’ve gone through a lot of trouble (“Why isn’t the field group working?”) because it wasn’t started.

```
MyFieldGroup:start(...)
MyFieldGroup:startMaster()
MyFieldGroup:stop(...)
MyFieldGroup:stopMaster()
MyFieldGroup:get(...)
MyFieldGroup:destroy(...)
MyFieldGroup:destroyMaster()

--For a single field
MyField.set(params)
```

Part 4 | Everything Else

Other Library Methods

In addition to the .VentGroup and .FieldGroup functions, there are a couple of other functions included in the CBEffects library.

Render

The .Render function sets the CBEffects render type. A value of “hidden” will make all VentGroups and FieldGroups with CBEffects be invisible and use no memory, while keeping errors from occurring. You can still call all functions for VentGroups and FieldGroups exactly the same, only everything is invisible. Calling things like “start”, “startMaster”, etc., will simply do nothing. It’s very useful for finding memory leaks.

Any value other than “hidden” will set CBEffects to normal.

```
-- To hide

CBE.Render(“hidden”)

--To set back to normal:

CBE.Render(“normal”)
```

DemoPreset

The .DemoPreset creates a VentGroup that’s simply a demo of a preset and starts it. No parameters are customizable. It’s for finding a preset to use. The only argument is the preset to demo.

```
local MyPresetDemo=CBE.DemoPreset(“smoke”)
```

DeleteAll

The .DeleteAll function destroys all existing CBObjects. Please note that you still must nil out the handle to them or memory leaks will occur.

```
CBE.DeleteAll()
MyVentGroup1=nil -- Still need to nil out handles
MyFieldGroup1=nil
```

Ending Notes

Believe it or not, I think I’ve finally finished with the gargantuan documentation. Make sure to notify me if you find something missing, a serious error, something you don’t quite understand, or anything to that effect on one of the two support links. Enjoy, and please don’t hesitate to ask for help!

Links

Support -
Support at the CoronaLabs Code Exchange
Support at the CoronaLabs CBEffects Forum Page (not recommended)

Other Links -
CBEffects Facebook Page
Gymbyl Coding Website
CoronaLabs Website