

Torque Game Builder – Particle Engine

Hierarchy

The Torque Game Builder (TGB) particle-engine is essentially split-up into two components. These are the “effect” (t2dParticleEffect) and the “emitter” (t2dParticleEmitter). An emitter is an object that actually emits particles in a configurable manner. An unlimited quantity of emitters can be created within an effect. An effect is essentially a container of emitters. Emitters cannot ‘see’ each other and therefore cannot alter each other's properties. An effect **can** alter emitters' properties by scaling them as a group.

As a practical example, let's assume we want to create a fire-effect. We want the effect to have three components: smoke, flames and sparks. The good thing is that the effect allows you to create these three separate components and keep them all together, handling them as a single object but still allowing individual emitters to be configured. We can go ahead and create three emitters, each of which generate a component of the fire-effect: smoke, flames and sparks.

If we want to tweak an aspect of the smoke, we can go into the effect and look at the smoke-emitter we created and tweak its configuration. What if we want to increase the particle-sizes for all emitters in the effect, not just a single one; how do we do this? It's actually really easy! We can do this because almost every configurable parameter in an emitter is duplicated in the container effect. How does this help us? Quite simply, the effect parameters scale all emitter parameters contained within it. This means that if we set the particle-size to x2, all particles in all the emitters become twice the size. This is a means to make effect-wide changes instantly.

As a note: not all parameters in the effect scale the emitters. The effect can directly control other emission properties if the respective emitters allow this. See below for more details.

The hierarchy for the fire-effect described above is shown below:-



particle-engine hierarchy

Torque Game Builder – Particle Engine

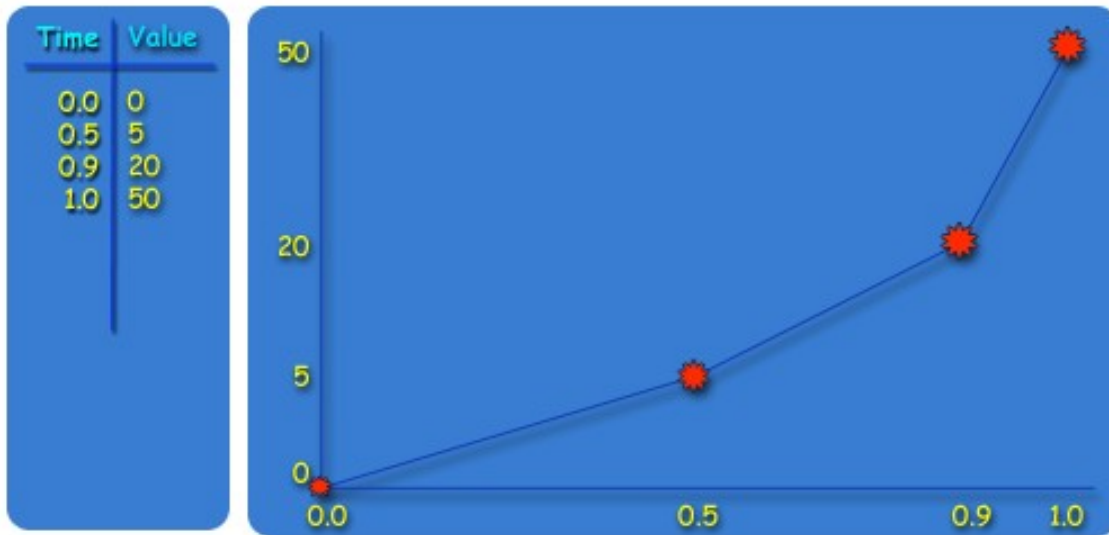
Fields

An effect or an emitter can be controlled via its fields. These fields are split-up into two groups.

The first group contains fields that simply contain values that are on/off, or have one or two values for things such as size etc. These values cannot be defined to change over time although they can be changed in script dynamically. These fields do things such as setting an emitter into “single-particle” mode or forcing the emitter to “track rotation” etc.

The second group, and the more powerful, contains fields which are called graph-fields. A graph-field is one which allows its values to be defined to vary over time, automatically in real-time. This is done by using a method called key-framing (a term used in animation). Each graph allows you to add **keys** to it. Each key is comprised of a time and a value. Entering multiple keys at various times/values will configure the field to change its output value over a period of time.

This is illustrated below:



graph-field visualisation

As you can see, the above graph-field has **4** keys defined. Assuming the time is in seconds, we can see that the field's value will change from 0 to 50 over a period of 1 second. If this field controlled the particle-size, then the particles would rapidly grow as it aged towards 1 second.

The particle-engine automatically linearly interpolates between the values. When the particle aged beyond 1 second, it would maintain its previous key-value, which in this case is 50; visualizing this, it would be a horizontal line on the above graph extending right from the fourth key.

It is important to note that key-times always progress in forward-time. That means that all key-times must be ahead of the previous one and no two keys can contain the same time although they can contain any valid value and repeat that value in multiple keys.

Torque Game Builder – Particle Engine

Timelines

Each graph-field is comprised of keys which are specified against a time-line, but what is this timeline? The answer depends on whether you're discussing the effect or the emitter. The quick answer is that timelines come in two varieties. The first timeline is associated with the age of the effect (playing time) and the second timeline is associated with particle-age. These two timelines differ slightly in their definition of time though. Effect-age is specified in **absolute-time** (in seconds) since the effect started playing whereas particle-age is specified in **normalized-time** (see below for more discussion on this).

Before I discuss any more details, you should try to remember that the graph-fields are all suffixed in such a way as to describe what type of timeline they refer to. These are:-

Graph-Field Type Suffix	Timeline Description	Used In
xxx_Scale	Effect-Age Timeline (<i>Emitter Value-Scaling</i>)	Effect Only
xxx_Base	Effect-Age Timeline (<i>Base-Value</i>)	Emitter Only
xxx_Var	Effect-Age Timeline (<i>Variation-Value</i>)	Emitter Only
xxx_Life	Particle-Age Timeline (<i>Value-Scaling Over-life</i>)	Emitter Only

Okay, so that all sounds quite complicated but in fact, it's rather simple. From now on, we'll use the same colouring as above to indicate a field in the **effect** or **emitter**. As an example and using the above graph, assume it is a graph-field in the **effect** and assume the value is a particle-scaling factor for all particles. What we would have defined then is the following:-

- Scale when effect has been playing for 0.0 seconds is 0.
- Scale when effect has been playing for 0.5 seconds is 5
- Scale when effect has been playing for 0.9 seconds is 20
- Scale when effect has been playing for 1.0 seconds is 50

This is pretty simple indeed and very powerful. Again, note that the key-times here are related to the time from when you start the effect playing and is **absolute-time**. If you restart the effect, the time is also reset and therefore the values picked from the graph change. As mentioned above; if the effect time goes beyond the defined keys in any graph-field then the value will stay at the last defined key, in this case 50.

Let us now discuss the emitter. The emitters are a little more complex because they contain graph-fields for both types of timelines but are easily identified by the prefix as described above.

Emitters contain "**_life**" fields which work on a different time-line than the one described above as the time is **not absolute time** but is **normalized-time**. What this means is that 0 is the start of particle-life, 0.5 is halfway through the particle-life and 1 is equal to the end of the particle-life. You cannot use time outside of the range 0 to 1. This may seem a little strange but the lifetime of particles can be configured separately and using normalized-time means that changing the absolute particle lifetime wouldn't result in having to change all the key-times. A major advantage!

Again, using the above graph as an example, assume it is a graph-field in the emitter and assume the value is an absolute particle-size. What we would have defined then is the following:-

- Size when particle starts (0.0) would be 0
- Size when particle is halfway through its lifetime (0.5) would be 5
- Size when particle is 9/10ths through its lifetime (0.9) would be 20
- Size when particle dies (1.0) would be 50

Note that the above doesn't describe how long the particle lives, just its size over its lifetime. All these fields are easy to identify: they are suffixed with "**_life**" as described above.

Torque Game Builder – Particle Engine

Graph-Field Types

As described above, there are four types of graph-fields, one of which is exclusive to the effect. Let's start focusing on a real field within the particle-engine as an example of how these different field-types work. For this example, we'll use the "speed" field which has all four types of graph-fields, three in the emitter and one in the effect. These are:

- "speed_base" (effect-age timeline) in the Emitter.
- "speed_var" (effect-age timeline) in the Emitter.
- "speed_life" (particle-age timeline) in the Emitter.
- "speed_scale" (effect-age timeline) in the Effect.

As indicated above, the first three types of fields for speed are all in the emitter. These are "base", "var" and "life". Let's examine each of these in turn.

"_base" is a field which is used as a base-value. In this case, it would define the base-speed for a particle in world-units/sec. This value (as indicated above) is an effect-age timeline so this field defines the base-speed for particles (as they are emitted) as the effect ages. Note that this **does not** change the base-speed of particles **after** they have been emitted, only as they are emitted (created).

"_var" is a field which is used as a variation on the "base" value. In this case, it would define a variance around the base-speed. The term variance means that a random value of the range specified would be used. This value is not directly added to the base but sits 'around' the base. A variance of 10 would produce a value in the range -5 to +5. This value (as indicated above) is an effect-age timeline so this field defines the variation on the base-speed for particles (as they are emitted) as the effect ages. Note that this **does not** change the base-speed of particles **after** they have been emitted, only as they are emitted (created).

"_life" is a field which is used to scale the resultant value of "base" + "variation" in realtime, during the lifetime of the particle. In this case, it would scale the speed. This value (as indicated above) is a particle-age timeline so this field defines the speed-scaling for particles as the particles age. Again note that this **does** change the size of particles **after** they have been emitted. Zero on the timeline here indicates the time when the particle is emitted (created) and one being its death.

"_scale" is a field which only exists in the effect and is used to scale the "base" + "var" used when particles are created. Note that this affects **all** emitters in the effect! This value (as indicated above) is an effect-age timeline so this field defines the scaling for speed for particles (as they are emitted) as the effect ages. Again note that this affects speed values for **all emitters** in the effect. Note that this **does not** change the base-speed of particles **after** they have been emitted, only as they are emitted (created).

So as you can see, there are two areas that these fields apply to. The first is the value used when the particle is emitted (created) and the second being the scaling applied during the lifetime of the particle. We'll continue to discuss the "_scale" field in the effect but I will emphasize that it is only typically used for performing effect-wide scaling. Most effects would be created without using these fields. They can be exceptionally handy when you want to use a base effect and want different scaled versions or perhaps you want to render it in a different resolution camera-view and you need to scale the effect up/down. Either way, it isn't typically used when creating particle-effects from scratch.

Torque Game Builder – Particle Engine

So choosing a value when a particle is emitted uses “_base”, “_var” and “_scale” and because these fields are all related, they all use the same timeline which is the effect's age. Scaling a value during a particle's lifetime uses “_life” and uses the particle's age.

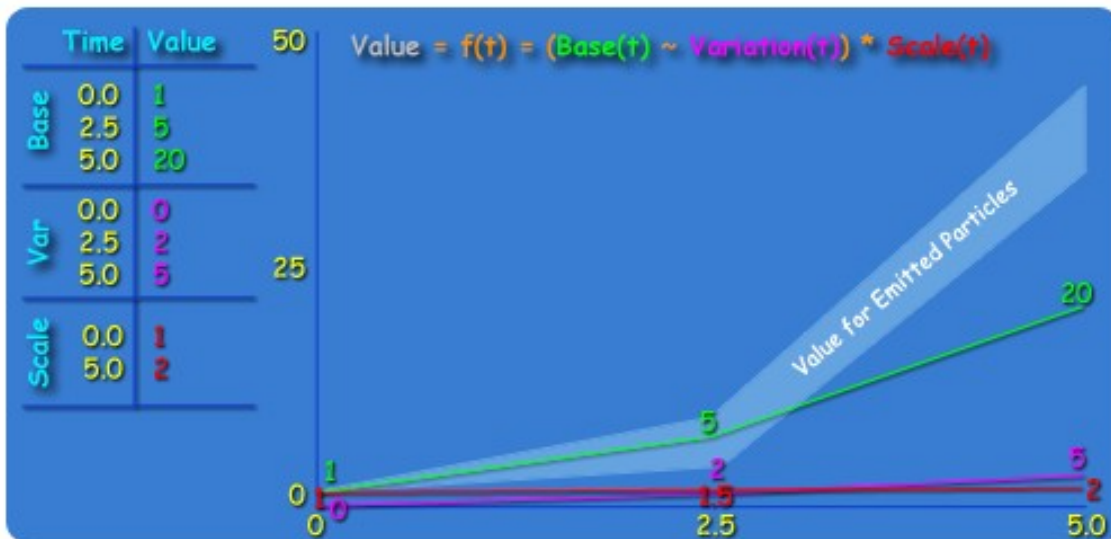
The formula for calculating the value used when a particle is emitted (created) is as follows:

$$\text{Value} = (\text{Base} \sim \text{Variation}) * \text{Scale}.$$

Assuming that at the current effects-age, the base is 10, the variation is 6 and the scale is 2, we would calculate the value as:

$$(10 + (-3 \text{ to } +3)) * 2 \text{ which ranges from } (10-3)*2 \text{ to } (10+3)*2 = 14 \text{ to } 26.$$

Here's an example (excluding “_life”) of particles being emitted:



As we can see here, as the effect ages to 5 seconds old, the value (let's assume in this case its speed) increases. Not only does it increase but there is a range of increasing values marked by the light-blue band. This band is created because of the variation field which adds a level of randomness to the value.

Taking the value at time 5.0, the value range is calculated as:

$$20 + \text{variance of } 5 \text{ } (-2.5 \text{ to } +2.5) = 17.5 \text{ to } 22.5$$

This range is then scaled by 2 to give a range of 35 to 45.

Now when doing this, you're not interested in the math so much as the way the effect looks so here's the way to use the fields. Set the “_base” value to the mid-point that you want the value to be. If you want a variance (randomness to the base), use the “_var” field. Quite often, you'll not even use the “_scale” field as this is in the effect and applies to all emitters (see note above).

Torque Game Builder – Particle Engine

What is “_life”?

Okay, so we understand that our “bread and butter” fields are just “_base” and “_var” and maybe “_scale” in the effect (for more advanced work) but what is “_life”? “_life” is an extremely important graph-field that differs from the others. Now that you’ve got your particles being created correctly, you probably want them to change whilst they are alive (during their lifetime). This is where the “_life” field comes in. As discussed before, the timeline for these fields is based upon particle-age and ranges from 0 to 1.

The “_life” fields is really simple to understand as it simply scales the value over the particles-life, identical to the “_scale” field with the difference being that the “_scale” field is in the effect and globally scales all emitter fields and also uses the effects-age, rather than the particles’. “_life” changes only the single emitter value it’s in.

Functionality

Okay, now that we know that in an emitter we’ve got “_base” and “_var” to choose an emission-time value and “_life” to change the value over the particles-life, as well as the ability to go to the effect and scale the value for all emitters, let’s move onto describing what the fields do, irrespective of how you can define keys.

Here are the graph-fields within the emitter...

Graph-Field	Field-Types in Emitter	Description
particlelife	base / var	How long the particle lives in seconds. This defines the time-time that “_life” works against.
quantity	base / var	Quantity of particles emitted each second. This value should be kept as small as possible to reduce the performance overhead of the particle-effect. It is typically more efficient to produce ‘bigger’ particles than it is to produce more particles when trying to “fatten-up” the effect.
sizeX	base / var / life	X component of size (If “fixedaspect” (See below) is used then this controls Y component as well). All sizes are in world-units.
sizeY	base / var / life	Y component of size (If “fixedaspect” (See below) is used then this is ignored).
speed	base / var / life	Speed here scales the particles velocity. A speed of one therefore produces no change. It is very important to understand that speed alone doesn’t produce any movement; the particle must have an initial velocity for speed to be useful. Initial velocity comes from “emissionforce” and “fixedforce” (see below).
spin	base / var / life	Spin is simply rotation and is specified in degree/sec.
fixedforce	base / var	Controls the magnitude of a fixed, constant force (angle specified elsewhere) applied to the particles. This can be used to simulate gravity or wind effects.
randommotion	base / var / life	Controls the magnitude of random motion applied to particles. The magnitude is a variance in world-units. The random motion is omni-directional and is added to the particles position after its velocity calculation meaning that it doesn’t change the particles actual velocity so if it is reduced, the particles velocity stays the same.
emissionforce	base / var	Controls the particles initial velocity when emitted. This can be changed with “speed”.
emissionangle	base / var	Controls the base-angle that the particles will be emitted at (in degrees).
emissionarc	base / var	Controls the arc around the chosen “emissionangle” (in degrees).
red	life	Controls the red-channel for the particle-colour. If the particle texture is white, this controls how much red is applied to the particle.
green	life	Controls the green-channel for the particle-colour. If the particle texture is white, this controls how much green is applied to the particle.
blue	life	Controls the blue-channel for the particle-colour. If the particle texture is white, this controls how much blue is applied to the particle.
visibility	life	Controls the alpha-channel for the particle-colour. This essentially controls how transparent the particle is and scales the particle-texture itself. A value of 0 makes the particle completely transparent whereas a value of 1 makes the particle opaque. Note that if the particle-texture already has an alpha-channel and is already transparent, then a value of 1 will make its transparency the same as the original texture. Essentially, the alpha value here is multiplied by any alpha-channel in the texture.

Coloured-regions only used to separate field-groups

That’s it! After all that discussion, we’ve only got a handful of actual fields but each allows very precise control over how the values change over time. Also note that there are additional fields in the emitter that are not graph-fields but set other options. Things such as setting an “imageMap” or turning “fixed-aspect” on or off. These will be detailed later.

Torque Game Builder – Particle Engine

Well that's the emitter, so what about the effect itself. Don't forget that "scale" fields will scale **all emitters** in the effect. Anyway, here's the list of fields...

Graph-Field	Field-Types in Effect	Description
particlelife	scale	See Emitter for description.
quantity	scale	See Emitter for description.
size	scale	See Emitter for description.
sizey	scale	See Emitter for description.
speed	scale	See Emitter for description.
spin	scale	See Emitter for description.
fixedforce	scale	See Emitter for description.
randommotion	scale	See Emitter for description.
visibility	scale	See Emitter for description.
emissionforce	base / var	See Emitter for description.
emissionangle	base / var	See Emitter for description.
emissionarc	base / var	See Emitter for description.

The "odd" ones here are the "emissionXXX" fields. These **don't** scale all the emitters, they do something else that can be very useful. There is an option in the emitter ("*effectemission*" – see below) that tells the emitter to ignore its own "emissionXXX" fields and use the ones in the effect. This means that individual emitters can choose to either have their own unique emission properties or use the parent-effect emission properties. An example of this being used would be if you have several emitters that all emit in the same way but a couple of them need to do something specific. This allows you to change all the common emissions at the effect-level but customize individual ones per emitter. The default is to use the effects emission so you'll want to turn this off if you need to configure the emitter so that it has its own emission controls (see below).

Using Graphs

Now that we've seen a complete breakdown of what graphs are we can begin to look at the methods involved to modify their content. Note that this applies equally to effects as it does individual emitters as both objects contain these graph-fields and use the same methods to modify their content. Anyway, here's a list of the functions available to modify a graph:

Script Function (Ref. Doc)	Description
selectGraph()	Before you can begin to edit any graph-field, you must select it. If you select a graph-field, it will stay selected for the effect/emitter until another is selected. When you have selected a graph-field, all the following functions (see below) work on it. This is convenient and removes the need to specify the field in all the functions below.
addDataKey()	Adds a data-key to the graph. A data-key is comprised of a time and value combination. If a key at the same time already exists, it is overwritten with the new key.
removeDataKey()	Removes a data-key from the graph. Note that you cannot remove the data-key at time-zero! Note that data-keys are referenced by their key-index, not time or value.
setDataKeyValue()	Changes the value of an existing data-key.
clearDataKeys()	Removes all data-keys, leaving a data-key at time-zero with the graph-fields default value.
setTimeRepeat()	This option allows you to change the time-scaling for retrieving values from the graph. If data-keys are defined up to 10 seconds and the time-repeat is set to x2 then the final key will be reached in 5 seconds scene-time. The reason this function is called "repeat" rather than "scale" is because it is primarily used on "life" fields. Setting "x2" on these fields effectively repeats the fields twice during the lifetime of a particle whereas non "life" fields don-repeat. If a particle had its red-channel set to ramp from 0 to 1 during its life and "x4" was set for repeat, the particle would ramp from 0 to 1, 4 times during its life.
setValueScale()	This options sets a scale for all values retrieved from the graph. This is a convenient way to scale all the values coming from a graph without the need to change existing key values.
getDataKey()	Retrieves a data-key time/value combination.
getDataKeyCount()	Retrieves a count of data-keys. This can be used to enumerate the list of data-keys in combination with "getDataKey()" above.
getMinValue()	Each graph-field sets bounds on the minimum value that can be used in any key. This is fixed by the system to a valid value. You can retrieve the minimum value using this function.
getMaxValue()	Each graph-field sets bounds on the maximum value that can be used in any key. This is fixed by the system to a valid value. You can retrieve the maximum value using this function.
getMinTime()	Each graph-field sets bounds on the minimum time that can be used in any key. This is fixed by the system to a valid value. You can retrieve the minimum time using this function.
getMaxTime()	Each graph-field sets bounds on the maximum time that can be used in any key. This is fixed by the system to a valid value. You can retrieve the maximum time using this function.
getGraphValue()	This function allows you to retrieve a calculated value from the graph at a specified time along the graph. You should specify a time within the range of "getMinTime()" and "getMaxTime()" (see above). This isn't used by the particle-system but can be useful for debugging purposes or other general use.
getTimeRepeat()	Retrieves the time-repeat setting (see "setTimeRepeat()" above)
GetValueScale()	Retrieves the value-scale setting (see "setValueScale()" above)

Torque Game Builder – Particle Engine

Grunt Work...

I can hear you say, “That’s great but how to I actually use these methods to manipulate keys in script?” No problem, let’s work through a few examples of doing just that.

Let’s assume we have the “fire effect” (described earlier) and that we want to create four keys at times, “0.0”, “0.5”, “0.9” and “1.0” in the “speed_life” graph-field of the “sparks” emitter. That sound like it’s going to be quite complicated but it actually isn’t so let’s just show the whole thing and then go through it piece by piece:

```
// Create a blank effect.
%effect = new t2dParticleEffect() { scenegraph = mySceneGraph; };

// Load our fire-effect from disk.
%effect.loadEffect( "fire.iff" );

// Find the "sparks" Emitter.
%emitter = %effect.findEmitterObject( "sparks" );

// Select the "speed_life" graph-field.
%emitter.selectGraph( "speed_life" );
// Clear any existing Keys.
%emitter.clearDataKeys();
// Add our keys.
%emitter.addDataKey( 0.0, 0 );
%emitter.addDataKey( 0.5, 5 );
%emitter.addDataKey( 0.9, 20 );
%emitter.addDataKey( 1.0, 50 );
```

The first two commands are pretty simple; we create a blank particle-effect and then load-up our pre-baked “fire” effect. Next we use the “*findEmitterObject()*” to get hold of the emitter we’re interested in, in this case the “sparks” emitter. Note that we could’ve selected the emitter by its index (if it didn’t have a name) using “*getEmitterObject()*” but this can change if the emitter order changes so it’s always preferable to access emitters by name. The emitters in this example effect are “smoke” (index#0), “flames” (index#1) and “sparks” (index#2) so we could’ve easily used:

```
...
// Get the "sparks" Emitter by index.
%emitter = %effect.getEmitterObject( 2 );
...
```

Anyway, let’s move onto the interesting part of this example which is actually selecting and changing a graph-field. We start this process by ‘selecting’ a graph-field using “*selectGraph()*” This may seem a little strange but it actually saves lots of work! By selecting a particular graph-field, you’re telling TGB that when you use a command that affects graph-fields in **this emitter**, it applies to the one you selected. TGB will use this graph-field until you change it so you only need to select it once. This saves you from having to specify which graph-field you are configuring in each command.

With the graph-field selected, we act a little cautiously by clearing any existing keys in the graph-field. What this actually does needs a little explaining though. The first thing to explain is that TGB ensures that all graph-fields are always valid. To do this, TGB enforces the rule that there must always be a key at time “0.0” and that all values for keys must be within the allowed range for the graph-field in question.

With that knowledge, it should be obvious that when you use “*clearDataKeys()*”, TGB does indeed clear any data-keys in the graph-field but it always automatically adds a data-key at time “0.0” that contains the default value for the graph-field in question (as defined in the reference documentation).

Torque Game Builder – Particle Engine

The good thing here is that you don't have to constantly check if a data-key already exists at the time you want to add a key. TGB will simply overwrite the key if it exists, so if we were to use the following code...

```
// Add our keys.
%emitter.addDataKey( 0.8, 0 );
%emitter.addDataKey( 0.8, 1 );
%emitter.addDataKey( 0.8, 2 );
%emitter.addDataKey( 0.8, 3 );
%emitter.addDataKey( 0.8, 4 );
%emitter.addDataKey( 0.8, 5 );
```

... which repeatedly adds a key at time "0.8", it would result in the value "5" defined at time "0.8". This simple feature means that you can just add a data-key to time "0.0", knowing it already exists.

Anyway, back to our example; now that we've 'cleared' the graph-field, we can proceed to add four keys to it with:

```
// Add our keys.
%emitter.addDataKey( 0.0, 0 );
%emitter.addDataKey( 0.5, 5 );
%emitter.addDataKey( 0.9, 20 );
%emitter.addDataKey( 1.0, 50 );
```

Well that was too easy so let's manipulate these keys a little so we can see how it's done. What we want to do now is remove the final key at time "1.0" and change the value at time "0.9" to "30". We do this by extending the original code with:

```
// Remove the final key.
%emitter.removeDataKey( 3 );
// Change the key at time "0.9".
%emitter.setDataKeyValue( 2, 30 );
```

As you can see, we use the "*removeDataKey()*" method but what is that curious value of "3"? It's actually the key-index, identical to the indexes that can be used for addressing emitters. The keys we've got are "0.0" (index#0), "0.5" (index#1), "0.9" (index#2) and "1.0" (index#3) so we want to remove index#3. This may seem a little strange but you can use the methods "*getDataKey()*" and "*getDataKeyCount()*" to step through the keys for any graph-field.

Now that we've removed the final key, we want to simply set the key at time "0.9". In this case, we used the method "*setDataKeyValue()*" to directly change the value of a key. Again, we're using the key-index notation to address a key.

Let's elaborate a little on the use of a key-index as it may seem a little complicated. In the above example, we use the method "*setDataKeyValue()*" but we could've equally specified the change using the time "0.9" like so:

```
// Change the key at time "0.9".
%emitter.addDataKey( 0.9, 30 );
```

This would've made much more sense as we know that adding a key, where a key already exists, just overwrites that key and we also knew the time we wanted to change, but what if we didn't know the time and we wanted to go through all the keys and set them to a specific range of values?

Torque Game Builder – Particle Engine

Let's do just that:

```
// Get Key-Count.
%keyCount = emitter.getKeyCount();
// Step through all keys.
for ( %n = 0; %n < %keyCount(); %n++ )
{
    // Get 'old' Key using its index.
    %oldKey = %emitter.getDataKey( %n );

    // Change Key to some arbitrary value.
    %emitter.setDataKeyValue( %n, %n*10 );

    // Get 'new' Key using its index.
    %newKey = %emitter.getDataKey( %n );

    // Dump to Console.
    echo( "Key-change from" SPC %oldKey SPC "to" SPC %newKey );
}
```

As you can see, we start by getting the count for the number of keys in our currently selected graph-field. We can then step through the keys and use “*setDataKeyValue()*” to change each. In the example we use “*getDataKey()*” to retrieve the key both before and after we change the value. We can then dump the details of the key-value changes to the console.

You don't always have to enumerate the list to find key-indices though; TGB returns you the index to a key when you add it like so:

```
// Add our keys.
%keyTime10 = %emitter.addDataKey( 10.0, 0 );
%keyTime20 = %emitter.addDataKey( 20.0, 5 );
%keyTime30 = %emitter.addDataKey( 30.0, 20 );
%keyTime40 = %emitter.addDataKey( 40.0, 50 );
// Dump to Console.
echo( %keyTime10 SPC %keyTime20 SPC %keyTime30 SPC %keyTime40 );
// Outputs...
"0 1 2 3"
```

Note that the key-indices are time-ordered. It's also interesting to note that you don't have to add keys in any specific timewise order either, so let's do the above but in reverse order:

```
// Add our keys.
%keyTime40 = %emitter.addDataKey( 40.0, 50 );
%keyTime30 = %emitter.addDataKey( 30.0, 20 );
%keyTime20 = %emitter.addDataKey( 20.0, 5 );
%keyTime10 = %emitter.addDataKey( 10.0, 0 );
// Dump to Console.
echo( %keyTime10 SPC %keyTime20 SPC %keyTime30 SPC %keyTime40 );
// Still Outputs...
"0 1 2 3"
```

You should note that you can change all effect/emitter fields whilst the effect is playing, there are no restrictions. Leave an effect playing and tweak anything you want! Well that pretty much covers most of the basic things you'll want to do with graph-fields. There is, of course, much more functionality but we'll let you discover that for yourself.

Torque Game Builder – Particle Engine

Enough Graphs Already!

Now that we've discussed the graph-fields, we just need to discuss the other options available to the emitters. These are generally simple options to control emission or provide supplementary parameters to the graph-fields above. These are:

Emitter Option	Script Function (Ref. Doc)	Description
Name	setEmitterName()	Sets the emitters name. This is one method of retrieving the emitter from the effect in addition to retrieving via the emitters index. (See "findEmitterObject()" in ref. doc). Note that it is not necessary for the emitters to have names, they are a convenience only.
Type	setEmitterType()	There are four emitter types which control the region at which particles are created. Note that currently the area, which the types are based upon, uses the "effects" size e.g. "12dParticleEffect::setSize()". These types are:- <i>"Point"</i> – Point at the effects center. <i>"LineX"</i> – Line defined across effects X-axis. Y position is effects center. <i>"LineY"</i> – Line defined across effects Y-axis. X position is effects center. <i>"Area"</i> – Whole area of effects size.
Visible	setVisible()	Controls whether the emitter is showing particles. NOTE:- Unlike other TGB objects, making an emitter invisible, destroys any existing particles and stops the emitter from creating any more.
Orientation	setParticleOrientation()	There are three orientation types which control the orientation of particles when they are created and/or during their lifetime. Each option has further parameters which can be setting using other calls, detailed below. These orientation types are:- <i>"Aligned"</i> – Particles have their "up" direction aligned to the initial particle direction. Other options are "alignangle" and "keepaligned". <i>"Fixed"</i> – Particles are aligned to a fixed angle. The fixed angle is specified using "fixedangle". <i>"Random"</i> – Particles are aligned to a random angle. This random angle is specified using "randomangle" and "randomarc".
alignangle	setAlignAngleOffset()	<i>This option is used when the "orientation" mode is set to "aligned".</i> It allows a fixed-angle offset to be added to the alignment function. Because the align-mode orientates the particles such that the initial "up" direction is facing the direction of particle travel and that the particle-texture used may be orientated in a different direction, you may wish to rotate the particle by a fixed-offset.
keepaligned	setAlignKeepAligned()	<i>This option is used when the "orientation" mode is set to "aligned".</i> It forces the particle to stay aligned to its current direction during its lifetime. This will override any other settings such as "spin". Adding random-motion can cause the direction to change, this will keep the particle aligned to that new direction. In "aligned" mode, with this option off, the alignment applied when the particle is created.
fixedangle	setFixedAngleOffset()	<i>This option is used when the "orientation" mode is set to "fixed".</i> It indicates an angle for initially orientating the particle. Unless "spin" is used, the particle will stay orientated in this direction.
randomangle	setRandomAngleOffset()	<i>This option is used when the "orientation" mode is set to "random".</i> It indicates a fixed angle which the "randomarc" will be based-upon.
randomarc	setRandomArc()	<i>This option is used when the "orientation" mode is set to "random".</i> It indicates a random arc-range around the "randomangle". This works identically to the idea of "base" and "var" with "randomangle" being the "base" and "randomarc" being the "var".
imagemap	setImageMap()	This specifies the imagemap/frame used for the emitter particles. This will disable the "animationname" option.
animationname	setAnimationName()	This specifies the animation used for the emitter particles. This will disable the "imagemap" option.
fixedaspect	setFixedAspect()	This option controls how the graph-fields "sizex" and "sizey" (see above) are used. If this option is active, the "sizex" graph-field is used for both the X and Y axis ("sizey" being ignored), essentially forcing fixed-aspect sized particles. With this option disabled, "sizex" and "sizey" act independantly allowing non-fixed aspect sized particles.
pivotpoint	setPivotPoint()	This sets the pivot-point used to orientate the particle, particularly useful when the particle is spinning. The offset is from the top-left of the texture. "0.5 0.5" is the center of the particle, "1 1" is the bottom-right. Values greater than 1 can be used to create interesting rotational dynamics for particles, especially when they move.
useBlending srcBlend dstBlend	setBlending()	This option controls the blending mode. When blending is active, the source/destination blending factors are used for all particles generated by this emitter.
intenseparticles	setIntenseParticles()	This option overrides the blending mode (see "setBlending()" above). When active, the particles will render such that their intensity accumulates to white. This can produce a "bright" or "warm" glow effects. Note that particles rendered in this mode cannot be seen on a white background. Technically, this option activates blending and uses blending factors of "src=SRC_ALPHA" and "dst=ONE" so it's just a shortcut to a common blending option; obviously the same can be achieved by using the standard blending control.
singleparticle	setSingleParticle()	This option forces the emitter to only create a single particle. The particle is always created as a "point" type. has a speed of zero (doesn't move) and lives infinitely (until the effect is stopped). This can be useful when you want to create a single static effect using a single particle. Good for adding corona/glows to effects.
fixedforce	setFixedForceAngle()	This options controls the angle used for the graph-field "fixedforce" (see above) that provides the ability to apply a constant force to particles.
attachposition	setAttachPositionToEmitter()	This option attaches any particles to the emitter position (which is fixed at the effects position). Typically, if an

Torque Game Builder – Particle Engine

		emitter is moving, particles created move independantly. When this option is active, they move relative to the emitter. Note that this doesn't affect the particles "orientation".
attachrotation	setAttachRotationToEmitter())	<i>This option is used when the "attachposition" mode is active.</i> Typically, particles rotate relative to the world such that zero is "up". When this option is active, the particles additionally rotate around the emitter. Note that this doesn't affect the particles "orientation".
rotateEmission	setLinkEmissionRotation()	This option will rotate the emission direction (see "emissionangle" and "emissionarc") to that of the effect. Unlike "attachrotation" that depends on also using "attachposition", this only rotates the emission and not the whole particle-system (all particles that are active). Therefore, only the emission direction is rotated. With this option active, rotating the effect results in a relative emission rotation as well. Note that all rotations are cumulative: using "attachrotation" and "rotateemission" results in the rotations being used together).
effectemission	setUseEffectEmission()	This options controls whether the "emissionXXX" graph-fields are used from the emitter or the effect. This is described above in the effects-fields.
firstinfrontorder	setFirstInFrontOrder()	This option controls the rendering order for particles. With this option active, the first particles to be created are in-front. Another way of saying the same thing is that with this option active, the oldest particles are in-front with newer particles appearing behind the older ones. With this option inactive, the reverse is true. Note that this option is only really useful when particles are not transparent but completely opaque. Also, there is no performance penalty for either option e.g. no particle sorting is taking place.

Global Tweaking...

Now that you've seen all the functionality that's available from configuring the graphs, it's time to mention one last thing that you may find useful. It's a simple thing to get lots of effects and therefore particles up and running but one thing that you'll come across is the ability to bring your computer to its knees by having too many particles active. Worse, you may need to tweak the number of particles up or down dependent upon the computer your game is running on!

As you can see from above, you can scale each effect's "quantity_scale" graph to achieve this but if you've got lots of effects, it can be awkward to do this. To enable you to make global changes to the number of particles created, TGB provides the following preference:

```
"$pref::T2D::particleEngineQuantityScale"
```

This variable will scale the number of particles created by effects. A value of "0.5" will half the number of particles created. The default is "1.0" which doesn't change the quantity of particles created.

Collisions...

A feature that is most definitely advanced is the ability for particles to collide with objects in the scene. The following description assumes that you already know how to set up collision-detection on scene-objects.

Assuming you have already set up the effect-object to collide with objects in the scene, all you need to do to get the actual particles to collide rather than the effect-object is to issue the following command:

```
// Turn-Off "Effect" Collisions and use "Particle" collisions instead.  
// NOTE:- This option is "true" by default which causes the "effect" to collide.  
%effect.setEffectCollisionStatus(false);
```

After turning effect-based collisions off using the above command, all collision-detection will be particle-based. The result here would be that particle-effect objects **will not** collide with anything but their particles will. So how do you configure what objects these particles collide with? Easy, you've already done it: simply use the standard collision-methods against the "effect" object. In other words, the effect collision-settings are used to determine what its particles collide with. This method removes the need to have an alternate set of collision calls specifically for particles.

Torque Game Builder – Particle Engine

Now that you know how to activate particle-based collisions as opposed to effect-based collisions, there are a number of key differences that you should be aware of.

Only a specific set of collision-responses are available. If any unsupported collision modes are selected, no collision-response will occur but a collision-callback will still happen.

Only the following collision responses are available:

- Clamp
- Bounce
- Sticky
- Kill

Finally, when a collision occurs the following script will be called:

```
// "Particle" collisions Callback.  
// NOTE:- The standard "onCollision()" callback isn't processed!  
function onParticleCollision( %this )  
{  
}
```

As you can see, the callback is not the standard callback of "onCollision()". You'll also note that the only detailed provided is the effect-object itself. No hit-object, collision-normal or time is provided. The reason for this is because particle collision quantities can be huge and providing all this information in a single callback or indeed producing a callback for each and every particle that collides would soon bring any system to its knees!

Keep in mind that particle-collisions are currently meant for looks only and are not meant to provide detailed collision feedback. The above collision-callback is provided to at least provide a signal as to whether a collision occurred which could be used to manipulate the effect somehow.

Torque Game Builder – Particle Engine

Back to basics...

This document has taken you on a grand tour of the TGB particle-engine and as you can see, there's a wealth of configurable parameters but in the end, most of this editing will be done in the particle-editor. Most of the time you'll only do basic things when actually 'using' a particle-effect in your application so let's spend a little time giving that kind of basic functionality some attention.

The stages of using a particle-effect are roughly split-up into four sections; create, load, play and stop. Let's do the first two together as they're so easy to describe:

```
// Create a blank effect.
%effect = new t2dParticleEffect() { scenegraph = mySceneGraph; };
// Load our fire-effect from disk.
%effect.loadEffect( "fire.eff" );
```

As you can see, it's pretty simple to get to this point. The `loadEffect()` method will load a complete particle-effect from disk and get it ready to go. These disk effects are typically saved using the particle-editor but can equally come from 'manually' created effects using `saveEffect()`. You could, for instance, load an effect created with the particle-editor, tweak some settings in script and then save the effect, perhaps with a different name, back to disk, it's totally up to you.

Now that we've got a particle-effect object, all we really want to do with it is to play the effect. We do this with the following:

```
// Play our effect.
%effect.playEffect()
```

That's it! You should note that it is at this point when the timelines for graph-fields start. Next we want to stop the effect. We do this with the following:

```
// Stop our effect.
%effect.stopEffect()
```

Again, that's it. This is too easy so let's discuss some of the finer-points of handling particle-effect objects. The first thing to note is that the effect-object is like any other TGB object in that when you've finally finished with it, you should `safeDelete()` it (see reference documentation). The method `stopEffect()` tells the effect to stop creating particles, it doesn't destroy the effect.

Torque Game Builder – Particle Engine

The stopping of effects raises an interesting dilemma. When we stop the particle-effect, what do we do with the particles that are still alive? Do we remove them? Do we wait for them to die naturally? If we remove them immediately, we'll see particles simply disappear which looks unnatural. We'd want them to die naturally but we don't know how long to wait; help!! The good news is that TGB allows you to choose and will handle it for you!

The “*stopEffect()*” has two optional parameters that allow finer control of this situation like so:

```
// Stop our effect.
%effect.stopEffect( true, true )
```

As you can see, we've got two flags to choose from. The two flags are called “WaitForParticles?” and “KillEffect?” respectively.

When “WaitForParticles?” is true, the effect will immediately stop emitting particles but will wait for any existing particles to die naturally before finally signaling the effect as stopped. This is extremely handy and the most typically used configuration. When this is false, particles will be removed immediately.

When “KillEffect?” is true, the effect will automatically be deleted when the effect stops. If you are ‘waiting for particles’, the effect will be deleted when all particles have died naturally.

So our call above would result in any particles currently active, when we stop the effect, to die naturally, immediately followed by the effect being destroyed. This is neat housekeeping as you can stop and forget the effect.

There is an additional option in the “*playEffect()*” method called “ResetParticles?”. This is very handy if you stop an effect and then proceed to play it again. In this situation, you may want any existing particles to be either cleared or allowed to die naturally as we did with the “*stopEffect()*” method. The difference here is that the effect will immediately start playing, creating new particles, whilst the older particles continue their life as normal.

This can be handy for “thruster” effects that are turned on/off. You would do this by starting/stopping the effect, maybe as the player hits the ‘thrust’ key. In this case, you'd typically want the effect to leave any existing particles alone so you'd use:

```
// Play our effect.
%effect.playEffect( false )
```

Note that the default is to clear particles when playing an effect.

Another option that is very handy in controlling the life of an effect and its particles is “*setEffectLifeMode()*”. All life-modes, with the exception of “kill” mode, can be configured in the particle-editor but it is very handy to understand exactly how it works. There are four ‘life-modes’ that an effect can use, these are “INFINITE”, “CYCLE”, “KILL” and “STOP”. Here are some examples:

```
// Set-up some life-modes.
%effect1.setEffectLifeMode( INFINITE );
%effect2.setEffectLifeMode( CYCLE, 5.0 );
%effect3.setEffectLifeMode( KILL, 5.0 );
%effect4.setEffectLifeMode( STOP, 5.0 );
```

All modes control what happens to the effect after a period of time. “INFINITE” states that the effect will continue to play forever. “CYCLE” will cause the effect to restart continuously, forever, in this case every 5 seconds. “KILL” will cause the effect to be destroyed, in this case, after 5 seconds. “STOP” will cause the effect to simply stop, in this case, after 5 seconds. All times are from the point when the effect is first started. Note that the “CYCLE”, “KILL” and “STOP” modes all wait for particles to die naturally (see above).

Torque Game Builder – Particle Engine

Although not strictly needed, it is interesting to note how each emitter controls the “area” in which it creates particles. Each emitter has a method “*setEmitterType()*”. The description for this method can be found above (and in the reference documentation) but it is not immediately obvious how the physical area is defined.

Let’s apply the following method to our effect (assume all emitters have type set to “area”):

```
// Set our effect size.  
%effect.setSize( "200 150" );
```

Setting the ‘size’ of an effect probably won’t do what you may first expect. Because effects don’t actually render anything, its ‘size’ doesn’t seem to control anything; it certainly doesn’t change the size of particles, there are graph-fields for those settings. So what does it do? Well, it’s actually quite simple to understand. The ‘size’ of an effect is used by all its emitters. When you set an emitter to “area”, “linex”, or “liney”, the physical area uses the parent effects’ size. It essentially sets the physical region that defines the “area” or “linex/y”.

The method above would set an area of “200 x 150” in the world. If the emitter was set to “area”, the particles would be created within an area of “200 x 150” at the effect’s current position. If the emitter was set to “linex”, the particles would be created along a line, centered on the effects current position, from -100 to +100 in the X-axis and 0 in the Y-axis. The “liney” type is identical to “linex” but along the y-axis.

And finally...

So we’ve taken the grand-tour of the TGB particle-engine and you may be thinking that we’ve covered pretty much everything but that isn’t the case. There is so much more to discover but thankfully, it’s all creative discovery because with all these configurable parameters at your disposal, you should be able to generate some completely amazing effects.

If you come up with some great effects and you don’t mind sharing then be sure to post them as a resource, maybe even a particle-effects pack. You never know, we may even include them in the SDK and make you famous. **J**

Good luck and enjoy the TGB particle-engine.