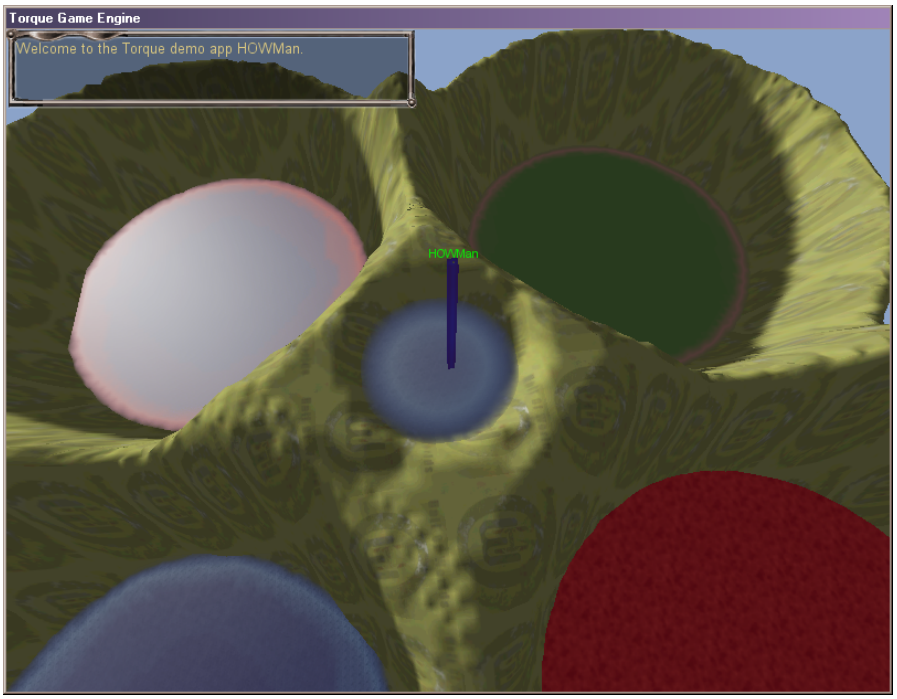
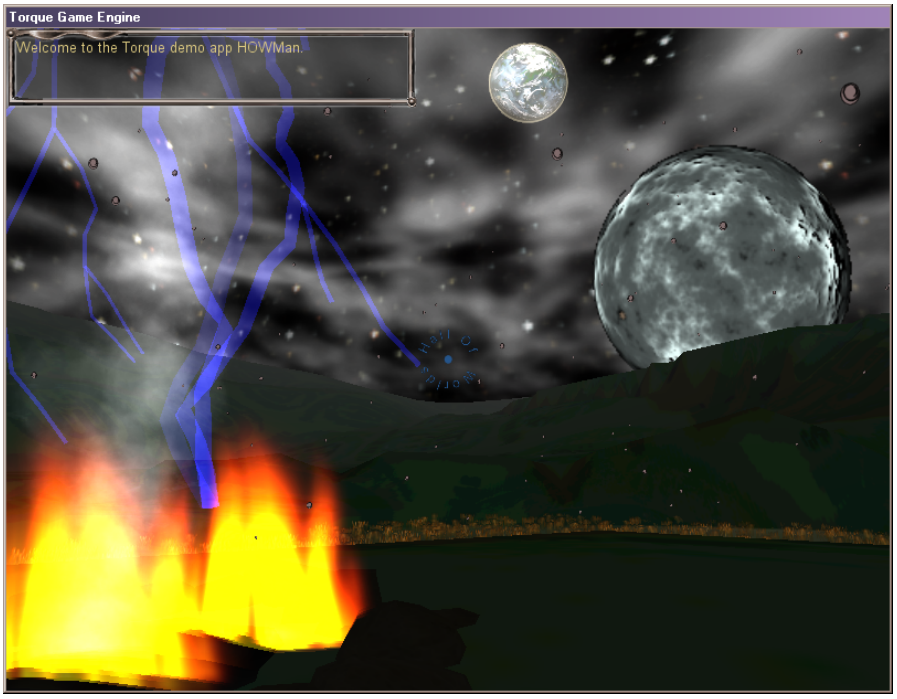
Section 3 – Tech School





Section 3 – Tech School 1

Adding Objects to Your World 3

Terrain 4
Water (Blocks) 13
Sky 22
Sun (Mission Lighting) 28
Precipitation & Lightning 32
Audio Emitters 37
Particle Emitter Nodes 49
fxShapeReplicator & fxFoliageReplicator 65
fxSunLight 76
Physical Zone 87
fxLight 90
Path 90
PathMarker 90
Trigger 90
Camera 90
SimGroups 90

Adding Objects to Your World

In this chapter, we will discuss how to add some of the most common mission objects and a few uncommon ones to your game world. For the most part, we will limit the discussion to basic usage. The goal here is to familiarize you with these objects and some of their attributes as well as to help you with any peculiarities. I won't necessarily cover every attribute of these objects in this chapter. Instead, an appendix is supplied, giving details on each object. It is assumed that you are familiar with the World Editor Inspector and Creator. If not, "Go directly to jail, do not pass go and collect $200", … or just go back to the beginning of 'Basic Training' and start there. When you're ready, come back and check this chapter out.

Terrain

In Torque, terrain is represented by a infinitely repeating heightmap. The heightmap itself is usually represented by a 256x256 full-color (24-bit) PNG image. The engine uses the single image as a home-tile, which is edge-blended and infinitely repeated in the world-plane. The default real-world measure of the home tile is 2048 meters on edge.

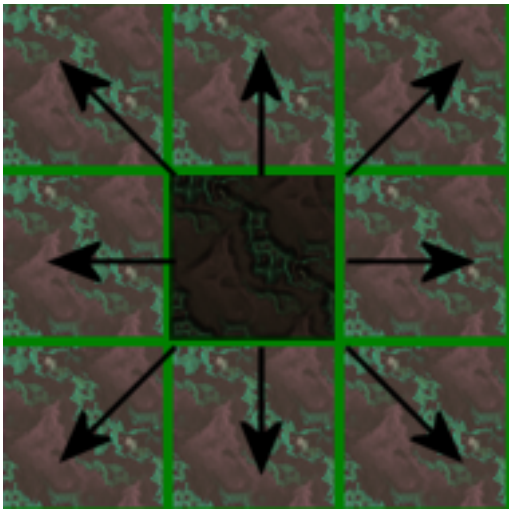


Figure 1 (TerrainTiling.png)

Terrain Features

Some of the terrain features that Torque supports are:

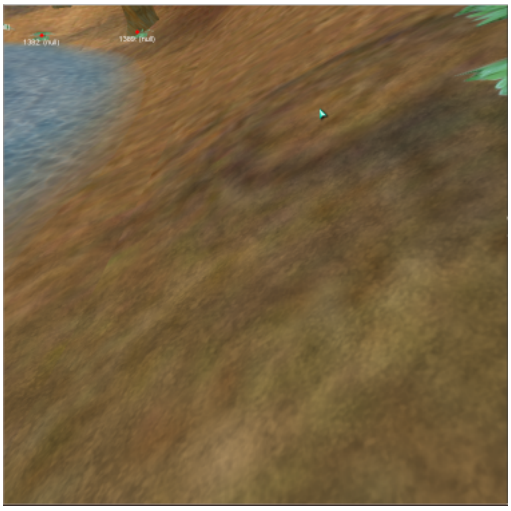
- A Detail Texture – A texture used to give more detail to locally visible terrain.
- Bump Mapping – The terrain supports emboss style bumpmapping, using a single source texture.
- In-Game Editing – With the Terrain Editor and the Terrain Painter, you can hand modify the shape and texturing of your terrain without leaving the game. This is described in Basic Training.
- Algorithmic Generation – The Terraformer provides a tool-set of algorithms for generating terrains. This is described in Basic Training.
- Algorithmic Painting – The Terrain Texture Editor provides a tool-set of algorithms for applying textures to the terrain. This is described in Basic Training.

- Alternate Sizing – Although it is advisable, one does not need to stick to a 2048 meter square home-tile.

- No Terrain – Finally, if not needed, the terrain can be removed entirely.

The Detail Texture

When you first start working with the terrain, it is easy to be overwhelmed and to miss an interesting yet important feature, namely the 'detail texture'. If you open up the inspector and select the terrain, you will see that there is field named *detailTexture* under the Media simgroup. This field provides the path to a texture which will be used to add detail to the local terrain. This additional texture is rendered once every 8x8 meters for N meters. Additionally, it is blended with the underlying textures with a ratio that falls off to zero at about 64 meters from the camera. Look at these screenshots to see the difference between terrain with and without a detail texture. I think you'll agree that the one with a detail texture is much nicer.



Terrain w/ Detail Texture

Terrain w/o Detail Texture

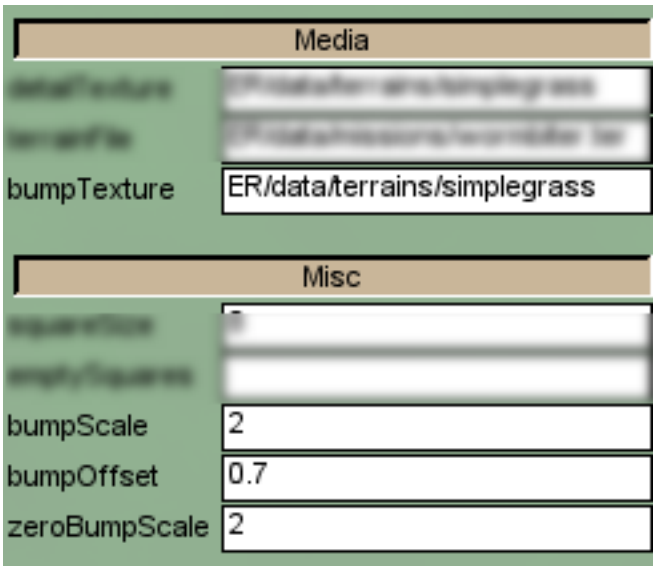
Great, right? Well, yes and no. Yes, because the terrain definitely looks better with a detail texture. No, because you can only have one per mission, which means all terrain in any single mission will have a fundamental sameness to it. For the most part, this is not a big deal and most players won't even notice. However, you need to realize that your choice of 'detail texture' can have a big impact on the visual quality of your terrain and you should probably count on having different textures for different levels/missions as this is a subtle way of creating distinct ambiances level-to-level.

Detail textures may be any size between 1x1 pixels and 512x512 pixels as long as they follow the standard rules for textures used by Torque.

Bump Mapping

For a long time, you could only get bump mapping by using Chris Weiland's patch resource "Terrain Bump Mapping". Over time Chris kept this patch up to date and made fixes based on feedback, but at some time he must have tired of the continuous need to keep it up to date and integrated it with the engine as a 'fix'. This fix provides the ability to enable 'emboss' style bump-mapping.

This features is controlled by four terrain parameters and a preference variable. It is simplest to edit the terrain parameters via the inspector:



- bumpTexture – Specifies a texture to use as the emboss map. Must follow Torque scaling standards for bitmaps, should be a mixture of blacks and whites, and it should tile. You must save the mission and re-load for this to take affect. The engine uses this texture to create the two textures required for embossing. One is the original, the second is the inverted original.

- bumpScale – Determines the how stretched the bump map texture is. In other words, small numbers cause the emboss map to cover a very small area, giving a more finely detailed bump mapping.

- bumpOffset – Is the offset between the two textures that make up the emboss bump map effect.

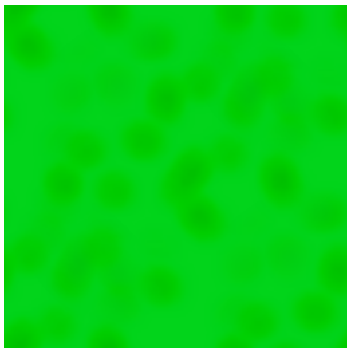
- zeroBumpScale – Controls the bump mapping radius. If you consider that Bump Mapping is only enabled within this radius (centered about camera), then it will be easy to understand that smaller values will cause the bump mapping to cease

near to the camera, while larger values will make it stretch further into the visible distance.

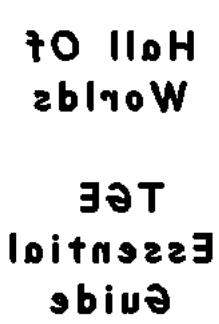
As noted, there is one preference variable.

- pref::Terrain::enableEmbossBumps – Allows you to disable this feature, which could be necessary on a slow machine or an older video card.

Because they say a picture is worth a thousand words, here are some bump mapping samples:



Base Texture



Bump0

Bump1

bumpTexture – Bump0

bumpScale – 3

bumpOffset –0.01

zeroBumpScale – 2

bumpTexture – Bump1

bumpScale – 3

bumpOffset –0.01

zeroBumpScale – 2

bumpTexture – Bump0

bumpScale – 8

bumpOffset –0.01

zeroBumpScale – 2

bumpTexture – Bump0

bumpScale – 3

bumpOffset –0.01

zeroBumpScale – 8

bumpTexture – Bump0

bumpScale – 3

bumpOffset –0.05

zeroBumpScale – 8

More about Terrain Painting

Although it might seem obvious, I'll say explicitly that the textures used to paint the terrain should be seamless. Why? Well, because the textures are repeated every *squaresize* meters. This means that with a default *squaresize* of 8, a painting texture repeats after only eight meters. Regardless, if your textures are not seamless it will be noticeable.

Alternate Terrain Sizing

Interestingly, when new folks start playing with Torque they soon realize that the terrain tiles. Then, after asking around they realize that the map is 'only' 2km x 2km. A percentage of these folks have in mind making some kind of game that would require a much larger terrain, say an MMORPG. They immediately focus on the problem of making the terrain bigger. In fact, if you are reading this I imagine that you might be one of those folks. Unfortunately, discussing the solutions in great detail is beyond the scope of this guide, but not to worry, there are resource available to deal with this 'problem'. Here are a few:

Torque Terrain Manager (Bryan Turner)

Bryan Turner has done the most complete work that I am aware of on a 'larger' terrain solution. In another lifetime, the Torque Engine did support multiple tiles of various sizes. In fact, must of the code is still in place or organized in such a way as to enable this. Bryan has modified said code to re-enable this feature as well as started work on integrating these changes into the editors. Unfortunately, as far as I can tell, this code was last updated some time mid-2002. Regardless, this is a great starting point if you have the programming skills required to integrate this into the engine. It is definitely better than starting from scratch.

Bryan Turner's work on the subject:

- http://www.fractalscape.org/TorqueTerrain
- http://www.fractalscape.org/TorqueTerrainManager.zip
- Thread: http://www.garagegames.com/index.php?sec=mg&mod=forums&page=result.thread&qt=4474

Modifying *squaresize*

A less robust method of modifying the terrain size is to change the terrain's *squaresize* parameter. This parameter can be edited in the inspector. It can be found in the terrain's 'Misc' simgroup.

But, what does changing the value do? If you will recall, the terrain heightmap is really nothing more than a two dimensional array of values. Furthermore, we normally represent heightmaps as a bitmap which (in Torque) is 256 pixels on a side. *squaresize* is a multiplier which specifies how many meters apart the pixels are in the heightmap. Sounds pretty simple right? In a sense it is. Legal values for *squaresize* are between 2 and 64 and are not 'strictly' limited to multiples of two, meaning you can have the following map sizes:

squaresize	Map (Home Block) Dimensions
2	512 meters squared
4	1024 meters squared
8 (default)	2048 meters squared
9	2304 meters squared
…	…
64	16 kilometers squared (this is 256 million square meters!)

This seems pretty good at first, but once we start playing around with it we start to see problem. The one most folks notice right away is 'water holes' At non-standard square sizes, water blocks will sometimes exhibit holes. That is square regions where there should be water, but no water is rendered. This is very annoying. Another problem is collision. Terrain collision is affected negatively by larger squaresizes. This can be so serious, that the player may actually fall through the terrain in some places. Finally, we run into the more subtle issues of memory usage and texture bandwidth. Varying *squaresize* modifies both memory usage and texture bandwidth associated with terrain rendering. I have personally noticed that a *squaresize* of 2 severely reduces FPS. So, given all these bad things, should you use this method? Sure, but only if you want to go up or down by a factor. Then, this is a good partial solution. I say partial because there are ways of solving the problems noted above. However, I'm going to leave this as an exercise for the reader, because generally I don't think folks really need big terrains, and you can just limit the size of your mission area if you want a small terrain.

In closing, there are two more things to mention. One, although you can use *squaresize* values that are not powers of 2, I don't suggest it. You will not be able to fix

the waterholes problem if you are using non-power-of-two *squaresize*. Two, if you do adjust squaresize, you need to adjust the position of the terrain block so it is centered. For example:

squaresize	correct position
4	"-512 -512 0"
8 (default)	"-1024 -1024 0"
16	"-2048 -2048 0"

Please read the next section and then, if you are not convinced that modifying your terrain size is a waste of time, feel free to follow up by reading these threads which discuss *squaresize* and waterholes:

- http://www.garagegames.com/mg/forums/result.thread.php?qt=9559
- http://www.garagegames.com/index.php?sec=mg&mod=forums&page=result.thread&qt=3166
- Big Dog Desmond Fletcher AKA 'The King of Tutorials' covers some interesting aspects of terrain, including *squaresize*, on his website: http://holodeck.st.usm.edu/vrcomputing/vrc_t/tutorials/

Big Terrains, Don't Do It!

I want you to stop and consider this small question, "How are you going to populate this very-large world you wish to make?" This might seem like a silly question, but let me assure you it is not. I once read something to the effect that the folks who made Tribes™ 2 were a bit worried about the map 'size' being a limitation, but quickly realized that it is very difficult to actually fill four square kilometers of space. In fact, most missions in Tribes™ 2 are much smaller than the maximum map size. OK, you may still be thinking, something like: "Yeah, but I can walk all the way across the map in like no time flat!" Point in fact, traveling at top speed, it will take you just shy of 2.5 minutes to walk from one side of the map to the other. This would make the Torque dude pretty darn fast. In fact, the default maximum speed for the character is 68km/hr. Consider that a normal human sprints as somewhere near 30 km/hr max. Why the big difference? Feel. It just feels too slow to make the character walk and run at normal human paces. Why is this even important? It is important for the following reasons:

1. You are going to have a heck of a time populating 4 square kilometers which is equivalent to about 400 square city blocks.
    - Note: There is no official dimension for a city block, but they average between 100 to 200 meters on end.
2. There are other solutions:
    - Just use the tiled terrain. Who is going to notice that it repeats if it take 2.5 minutes to run across it?
    - Slow the character down and tighten up spacing on objects. This is easier to do than increasing the size of the terrain. Guaranteed!

3. This is really going to hurt and you don't want to do it. OK, I'm not exactly telling the truth, but I can say that it is not simple to do this. Don't believe me? Fine! Now you can go try those ideas out. Sheesh! You can't blame a guy for trying to help…

No Terrain?

Certainly. If you wish to have a terrain-less mission, it is entirely possible. However, you'll have to edit the mission file to do this. Trying to delete the terrain from the Inspector just plain won't work. Simply open your mission file in any handy text editor, find the block named "TerrainBlock", and delete the entire thing. Viola! No terrain.

Water (Blocks)

After Terrain, Water is another hot forum topic. Fortunately, water has gotten a lot of attention from community Members like Big Dog Melv May. However, this additional attention has had the side-effect of making water 'seem' complicated to use. In reality, most options are just that, optional. You can place and set up water in just seconds, or if you want to go for a specific effect, you can spend hours tweaking the parameters. For brevity sake I will give the quick setup instructions first, then I'll cover the 'advanced options'

Basic Water (Mucho Rapido Setup)

OK, get your stop watch out. Start it. Now follow these instructions:

1. Start SDK
2. Open any mission (preferably Tech School).
3. Start the Mission Editor
4. Switch to the Creator tool
5. Switch to Free-camera mode and move the camera up a few meters
6. Look somewhere near your character.
7. Insert a new water block (Mission Object→Environment→Water)
8. Just Click OK for the dialog that comes up[1]
9. Switch to the Inspector tool.
10. Click on the water block.
11. Click the 'Expand All' button.
12. Change Media→*SurfaceTexture* to 'egt/data/water/howwater0'.
13. Make sure Debugging->*UseDepthMask* is NOT checked.
14. Set Surface→*surfaceOpacity* to 1.0
15. Set Surface→*envMapIntensity* to 0.0
16. Click Apply.

Done! Depending on the speed of your machine that should have taken about 60 seconds or less.

Water Features

Some of the water features that Torque supports are:

- Discrete Scaling – Because of the algorithmic nature of the water in Torque, water blocks are scaled in fixed increments. By default, this is 32 meters.

- Discrete Positioning – Again, as a byproduct of its algorithmic nature (and due to a sometimes overlooked terrain relationship), water is positioned in fixed increments. By default, this is 8 meters.

- Various Texture-based Effects:

  - Basic Surface Texture – Plain Jane base texture for water.
  - Shore Texture – An additional texture for shorelines.
  - Over and Under Environmental Maps – Static environmental reflections on the surface of water from above and below.
  - Specular Reflections – Simulates perturbed specular reflection from water surface.
  - Underwater Fog – Torque provides a static fog for when the camera is underwater.
  - Underwater Texturing – Under certain circumstances, up to two additional caustic textures will be rendered over the view.

- Waves – Torque supports basic sinusoidal waves.

- Viscosity and Density – These two real-world characteristics affect the character and objects that come into contact with the water.

- Predefined Water Types – Torque provides several predefined 'types' of water which give you various ready made effects.

- Flow – Torque can visually simulate flowing water.

- Distortion – If the above visual effects are not enough, you can use distortion parameters to make the water yet more realistic or un-realistic if you so choose

- Multiple Blocks – Lastly, you may have multiple independent blocks of water.

Advanced Water

Alright, unless you are just goofing around and learning the engine, it is pretty likely that you will want to make your water look a little more interesting. No problem there. Between the original features and the awesome upgrades done by Melv May (another big dog) Torque water can do some very cool things.

Position and Scale

Before we jump into the cool stuff, let's briefly discuss basic positioning and scaling. Unlike most objects, you cannot position or scale water blocks arbitrarily. Instead, the X and Y components of both position and scale are adjusted in discrete steps. Position <X, Y> is adjusted in steps of eight (8), and Scale <X, Y> is adjusted in steps of 32. For both position and scale, the Z parameter can be adjusted continuously.

On a side note, if you have been reading this guide straight through, you may recall that the default terrain *squareSize* is also 8. It is no coincidence that both position and scale are adjusted in multiples of *squareSize*. If you are going to play with non-standard terrain sizes, or if you are going to make modifications to the way water-blocks

work, you'll have to remember that terrain and water are closely related. Kissing cousins, you might say.

It is very important to note, that the Z parameter should NOT be zero. Most people make the mistake of not adjusting this parameter. Most of the time, this will seem OK, but if the camera will ever be under the surface of the water, then you must have a non-zero value for Z. More accurately, you must adjust Z parameter of a water block, such that the lower boundary of the water block is below the lowest point in the terrain, for all points in the terrain covered by the block. Why? Well, if you do not do this, you may encounter a strange bug where the water fog disappears at certain viewing angles. This can destroy any suspension of disbelief you have managed to accrue and it is very distracting.

The Various Textures (Media)

The water block has progressed greatly since the day Torque was first released. With this progression has come a profusion of new parameters, including a multitude of texture parameters. Fortunately, these parameters are simple to understand:

- *surfaceTexture* – This texture is used to define the base water layer(s). This texture is rendered in two layers, with one layer re-oriented at a 45 degree angle (about Z of course). This make the water more interesting.
- *shoreTexture* – We'll talk more about shorelines in a moment, but basically, Torque has the ability to render shorelines differently. When it renders the shoreline, it blends this texture with *surfaceTexture*, giving a nice visual effect.
- *envMapOverTexture* – If environmental mapping (see .Reflections and Specular Masks' below) is enabled, this texture is rendered when looking down onto the water from above. This represents an environmental reflection on the water's surface.
- *envMapUnderTexture* – As with *envMapOverTexture*, this represents an environmental reflection, but this is the texture you will see if looking up from beneath the water.
- *submergeTexture0* and *submergeTexture1* – These two textures are only used when *liquidType* is one of the Lava types (Lava, HotLava, or CrustyLava). These two textures are rendered perpendicular to the viewing plane. Additionally they are animated. A suggestion I was given, which I'll pass along, is to use two high quality (say 512x512 instead of the normal 256x256) grayscale caustics for these. Note: By making some simple changes to the source code, you can colorize the resultant output to the screen. (EFM – Discuss again in OJT)
- *specularMaskTexture* – This texture is used to make the surface of the water look as if it is reflecting light. Again, this should be some kind of caustic grayscale. The engine does take into account the position and elevation of the sun when rendering the specular effect. We'll discuss this more below in 'Reflections and Specular Masks'.

Makin' Waves

The water would not be very interesting if it were just a flat plane. Fortunately, Torque supports a wave feature. The bad part is it is a simple sinusoidal function. Nonetheless, it does a pretty good job and looks good for most purposes. If you wish to have waves, set the *WaveMagnitude* parameter to a non-zero value. Bigger values equal bigger waves. Note, it is best not to attempt to place two water blocks side by side if you are using waves. Because the algorithms for each block are calculated separately, you will get visible seams and discontinuities. Also note, there is one disappointing thing about waves. If your player is floating in water (see 'Sinking and Floating' below), the waves will not raise the player. That is the water motion does not affect the player's vertical position, nor will splash effects occur from water hitting a motionless player.

Sinking and Floating

You may be wondering about how to make a character float, or perhaps you would like to make the water more viscous, say like quicksand. Well, Torque supports to water parameters for these effects:

- *density* – The default water density is 1. Meanwhile, the default character density is 10. This means the character will sink upon entering the water. So, if you want the character to be more buoyant, you can adjust either or both parameters. Just remember the following rules:
  - water→*density* < player→*density* ➔ Player sinks.
  - water→*density* == player→*density* ➔ Player neither sinks, nor floats.
  - water→*density* > player→*density* ➔ Player floats.
- *viscosity* – In addition to choosing whether a character will float or sink in water, we can indirectly adjust how quickly this occurs by changing the *viscosity* of the water. A more thick fluid like, say honey, has a high viscosity, whereas plain water will have a low viscosity. By increasing this value, you create an effect where the player will require more time to float or sink.
  - Note: This also effects the player's ability to walk through water. If the *viscosity* of the water is high and the player is hip-high or further submerged, he will begin to slow appreciably while walking.

Liquid Types

The *liquidType* parameter was mentioned briefly above. Out of the box, Torque supports several water types. They are legacy types from the Torque 2 ™ days. Unfortunately, they are not all distinct any longer. Now you have three basic categories:

- Basic Water Types – All these behave similarly.
  - Water
  - OceanWater
  - RiverWater
  - StagnantWater
- Lava Types – These cause damage when the player enters the water-block, but currently damage is not applied any longer while the player is submerged. (Yes, this is a bug). Note: By default, all three lava types apply the same damage but

you can change this by editing their corresponding damage parameters, which can be found in the player.cs file. Recall, when the water type is one of the three lava's, *submergeTexture0* and *submergeTexture1* will be rendered if you have specified them
    - Lava – Damage parameter is $DamageLava.
    - HotLava – Damage parameter is $DamageHotLava.
    - CrustyLava – Damage parameter is $DamageCrustyLava.
- Quicksand – This is in a class by itself. I'm not sure exactly what this used to do, but now it behaves just like water, except that the underwater fog does not render.

For most purposes a *liquidType* of either Water or Lava will suffice.

Underwater Fog

So, what is this underwater fog? The engine employs a fixed color for water fog, which cannot be adjusted via script, yet. I say yet, because if you needed to adjust this you could just expose the parameter to the console with a little coding. (EFM – Discuss in OJT).

- Note: As of the time I am writing this part of the guide, the code to change underwater fog is located on line 900 of game.cc:
    - `glColor4f(.2, .6, .6, .3);`

I motioned it above, but just in case you missed it, the Z parameter of your water block's position should be non-zero if you intend for the player (camera) to travel below the water. If you do not properly adjust this (see 'Position and Scale' above), there will be times when the underwater fog fails to render.

Water Flow

So far, we've talked about how to make waves, but what about horizontal effects, like water flow? Torque supports this too. You can cause specific textures to translate over time, givingthe illusion of water flow. The following parameters are involved:

- *FlowRate* – If this value is non-zero, water flow will be enabled. The higher the value, the more quickly textures will translate. The following textures flow:
    - non-oriented *surfaceTexture*
    - *shoreTexture*
- *FlowAngle* – This parameter (in degrees) determines the direction of the translation. The following values demonstrate the direction of flow based on angle:
    - 0˚ – Textures will translate in the negative direction along the World X-axis.
    - 90˚ – Textures will translate in the negative direction along the World Y-axis.
- *SurfaceParallax* –When *FlowRate* is non-zero, the flow-rate of the oriented *surfaceTexture* is controlled by this value as follows:

SurfaceParallax	***surfaceTexture* **vs.** **oriented *surfaceTexture***
magnitude greater than 1	non-oriented *surfaceTexture* flows more slowly than oriented *surfaceTexture*.
magnitude equals 1	non-oriented *surfaceTexture* and oriented *surfaceTexture* flow at same rate.
magnitude less than 1	oriented *surfaceTexture* flows more slowly than non-oriented *surfaceTexture*.
magnitude equals 0	oriented *surfaceTexture* remains stationary.
negative values	oriented *surfaceTexture* counter-flows.

Water Distortion

In addition to supporting waves, and water flow, Torque supports a distortion feature. It is difficult to classify this effect, because by varying the distortion parameters, you can get wildly different effects. However, the basis for these effects are simply the stretching and squeezing of the *surfaceTexture*'s and the *shoreTexture*'s uv coordinates across a defined grid. The parameters involved are:

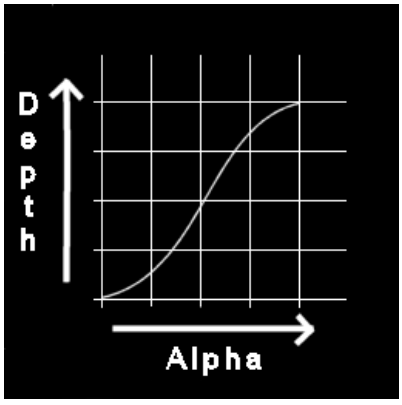
- *DistortGridScale* – You don't normally need to vary this from its default value, unless you have scaled your water. This allows you to adjust distortion such that the effect is the same between a large water block and a small water block. There are not set rules really. You'll just have to experiment.
- *DistortMax* – If this vale is not zero, distortion is enabled. Generally, the magnitude of this value should be less than 1 or the distortion behaves…strangely. Both positive and negative values are legal.
- *DistortTime* – As you might guess, this period of the distort function. It is inversely proportional to the distortion's rate of change. In other words larger values mean slower distortions and smaller values mean faster distortions. A value of zero (0) is illegal and will cause the texture rendering to fail gracefully.

Realistic Shoreline Rendering

We've mentioned the *shoreTexture* several times now, but avoided discussing how and when it is used. Melvin May modified the code to multi-texture the *shoreTexture* with the *surfaceTexture* based on the depth at that location and these parameters:

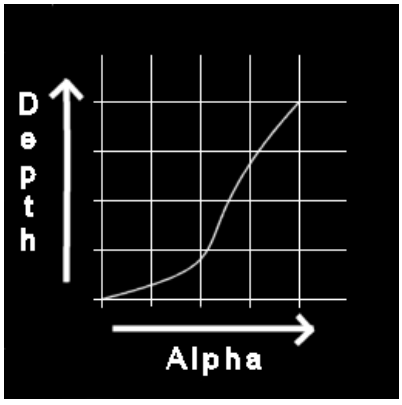
- *ShoreDepth* – Shore rendering is determined by a ray-cast at distinct points across the surface of the water block. The result of this ray-cast returns the distance between the top of the water and the terrain directly below that point on the surface. If this value is greater than or equal to *ShoreDepth* the engine is instructed to render the *shoreTexture*. If you choose to set this value to zero, the *shoreTexture* will not render at all.
- *MinAlpha*/*MaxAlpha* – As might be intuited, these two parameters determine the minimum and maximum alpha to use while rendering shoreTexture. This directly affects the multi-texturing equation involving the *surfaceTexture* and *shoreTexture*.
- *DepthGradient* – Controls the slope between *MinAlpha* and *MaxAlpha*. In older versions of the engine, Melv implemented this as a sigmoid function, but since version 1.2, it has been implemented using the (more involved) gama-correction function. This gives us the following depth vs. alpha curves:

Sigmoid (older versions)

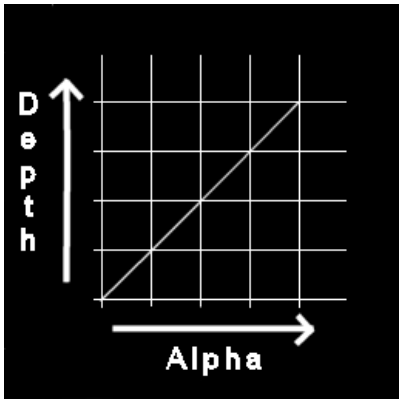


0 < *DepthGradient* < 1

Fast Fade-Out
Slow Fade-In

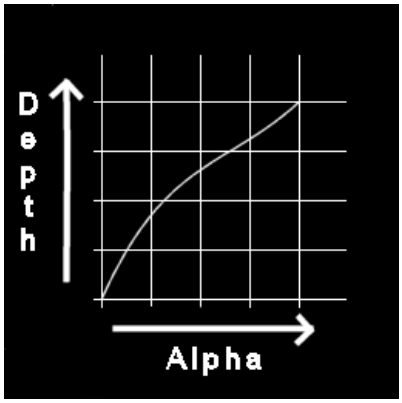


DepthGradient = = 1



1 < *DepthGradient* < 0

Slow Fade-Out
Fast Fade-In



Reflections and Specular Masks

Torque doesn't support real-time reflections, but it does support the next best thing, which is a static environment map. In fact, as noted above, it supports two maps. One for above the water and the other for below. In addition to being able to specify these two environment maps (using *envMapOverTexture* and *envMapUnderTexture* respectively) you determine their how they blend by adjusting the *envMapIntensity* parameter. Legal values are between zero and one.

In addition to environmental mapping, Torque support specular masks to simulate highlights. The specular mask is used to make the surface of the water shiny, that is, to provide interesting looking highlights. When you use a specular mask, the engine will render highlights, based on the texture you provide (*specularMaskTexture)*, the position of the sun, the elevation and inclination of the camera, and two additional specular parameters:

- *specularPower* – This determines how large an area is shiny. Lower values cause more of the specular map to be rendered, versus larger values which will tend to show just a spot of highlighting.
- *specularColor* – This can be used to change both the color of the resultant highlight and its intensity. This parameter takes a four-element vector "r g b a".

The *specularMaskTexture* should be a gray-scale caustic for a natural looking water highlight.

Texture Scaling

Two parameters have been provided to allow you to modify the scale of the *surfaceTexture* and the *shoreTexture* rendering. These are named *TessSurface* and *TessShore* respectively. Low values result in the textures covering large areas of water prior to repeating, whereas large values cause the textures to repeat over shorter distances. Some caution is in order when using these parameters. First, extremely small values may cause the textures to become distorted. Second, extremely large values can cause texture aliasing even when the camera is very near to the water. Just remember, if you cause your graphics card to have to down-scale the texture when the camera is near to the water, you are wasting your artists' time.

Tying Up Loose Ends

In addition to the water block parameters covered thus far, there a few additional one. First, there may be several under the Dynamic simgroup. You can remove all of these. None of these parameters are hooked to anything Torque 1.2 and beyond. The remaining parameters are:

- *rotation* – Water blocks cannot be rotated.
- *UseDepthMask* – Caution is in order for this parameter. You may crash the engine if you attempt to change this in the inspector or from the console. So, if you want to experiment, change the mission file directly. Simply stated, if you value is false, only the *envMapOverTexture* will be rendered on the top of the water. All other 'surface' textures will be disabled.
- *surfaceOpacity* – I considered discussing this above, but it is pretty obvious. This affects how opaque the combination *surfaceTexture* and *shoreTexture* is. That is it. Now, a value of zero is not transparent, just very translucent. However, a value of one is quite opaque. You'll have to adjust this meet your needs.
- *removeWetEdge* – EFM –TBD
- *AudioEnvironment* – EFM - TBD

Sky

In standard Torque, the sky is depicted by a sky-box. In addition to the six sides of the box, you may specify up to three textures for cloud layers and three separate colored fog layers.

Sky Features

Some of the sky features that Torque supports are:

- Configurable Sky Box – As noted above, the sky is represented with a sky box. It offers such features, as disabling the bottom texture, and render bans.
- Three Cloud Layers – With the standard Torque sky, you can have up to three clould layers, each individually configured.
- General Fog and Three Layers of Fog – In addition to the generalized fog supported by the Sky object, you can define three additional 'layers' of fog.
- Visibility Distance – They Sky object is the place you go when you want to modify maximum view distance.
- Wind – The Sky object owns and controls the wind vector, which is used by other mission objects.
- Environmental Map – It may seem strange, but when you are seeking the environmental map that is used on characters and objects with environmental mapping enabled, this is the place you go. It is part of the skybox's texture list.

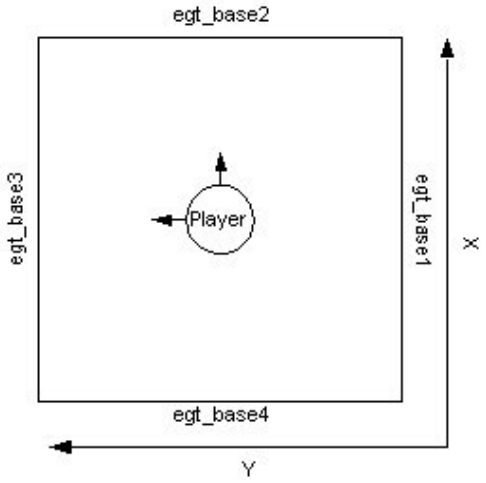
The DML File

As noted above, the DML file is the place you specify your skybox and cloud textures. The file itself can be placed anywhere you wish below the game root directory, since you can specify the relative path in the field: *materialList*. A sample file would look something like this (egt_base.dml):

```
egt_base1
egt_base2
egt_base3
egt_base4
egt_base5
egt_base6
env_map
layer0
layer1
layer2
```


In this example, egt_base1 .. egt_base4 represent the side textures, egt_base6 is the top of the box, and egt_base5is the bottom of the box. The first five textures are required for *useSkyTextures* true and *renderBottomTexture* false. The sixth texture is required if you choose to render the bottom texture of course.

So, what about the location of textures vs. the sides of the skybox? If we arbitrarily choose the World X-axis as our North-South axis, we get the following:



The next texture in the DML file is called env_map. This texture is used for any environment mapping applied to shapes. This texture is optional if you are not going to do any environment mapping and do not intend to have clouds.

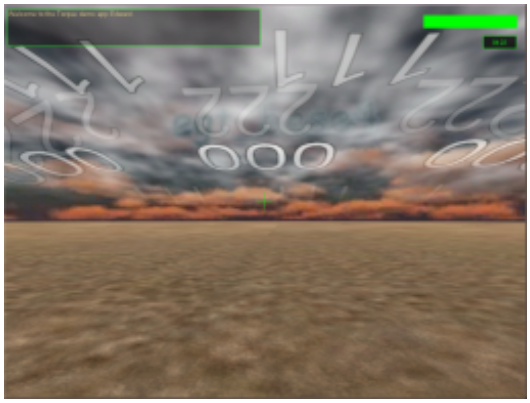
Finally, the last three textures in the DML file specify texture names for the cloud layers. The ordering of these textures has NOTHING to do with the cloud height. Cloud height is controlled by *cloudHeightPer[3]*. We'll talk more about this below

Please note, I've stated above that this or that texture is optional based on decisions you make. However, till you get rolling, I suggest that you always specify six textures for the sky box and one additional for the envmap. This way you won't run into any difficulties. Note also that the file is positional. So, for example, if you want clouds, you must have specified the seven prior textures, even if they are dummy textures that won't be used.

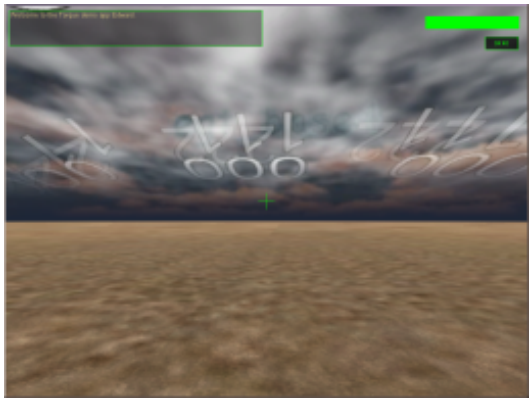
The Sky Box and Render Bans

"What exactly is a Render Ban," you might ask? Unfortunately I can't give a technical answer to this. Instead I'll give you an answer based on my perception of this feature. Let's consider the case where you have terrain in your mission. If you choose to enable render bans (*noRenderBans* == false), the side textures are variably rendered from the very top down to the level of the camera, but not below the level of the terrain. As we approach the point where rendering 'stops', the texture is blended with the background

color with a rapidly falling ratio. The visual effect is of a long horizon where the color fades to a uniform color (*SkySolidColor*). Personally, I think this gives a more pleasing (and realistic) effect vs. render bans disabled (*noRenderBans* == true). Compare these two images and decide for yourself:



noRenderBans == true



noRenderBans == false

It should be noted that the effect of this choice is especially evident from a height.

Clouds

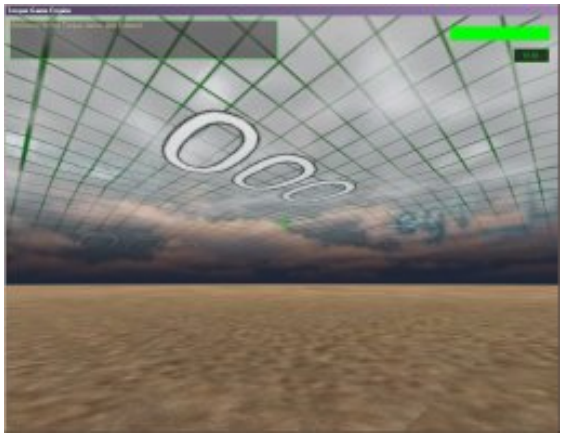
As mentioned above, the cloud layers are specified by textures eight, nine, and ten in the DML file. All cloud layers are optional.

cloudHeightPer

Texture eight corresponds to *cloudHeightPer[1]*, nine to *cloudHeightPer[2]*, and ten to *cloudHeightPer[3]*. These parameters (*cloudHeightPer*) are used to control the central height of the cloud meshes. The cloud meshes themselves are a sort of 9-sided 'hemisphere'. The *cloudHeightPer* parameter specifies the height of the upperplane of this 'hemisphere'. Here are some sample images to demonstrate the *cloudHeightPer* parameter:

One Texture: *cloudHeightPer* == 0.8

One Texture: *cloudHeightPer* == 0.5



One Texture: *cloudHeightPer* == 0.2

A value of 0.0 will cause the cloud mesh not to render, and values above 0.8 poke through the skybox causing visible artifacts.

Multiple Layers

In terms of viewing, Layer 2 is rendered first and Layer 0 is rendered last, meaning that Layer 0 will look like it is in front of 2 regardless of *CloudHeightPer*.

Cloud Motion

Cloud motion is described by two parameters. First, all clouds move in the same direction. This direction is specified by the (misnamed) parameter *windVelocity*. *windVelocity* is an x-y-x vector. The X and Y components control the direction of the wind and therefore the clouds. Putting a non-zero value in Z breaks the cloud renderer, don't do it. Finally, you can control the velocity of the flow with the *cloudSpeed* parameters. Yes, velocities can be negative, so clouds can counter flow.

Fog

Clouds are cool, but sometimes you need fog too, or instead of clouds. No problem. Fog is supported in Torque by a general fog, and by up to three cloud 'layers'. Unfortuntely, the Torque engine description did and perhaps still does say that Torque support volumetric fog. Strictly speaking, this is not true.

General Fog

The first type of fog supported affects visibility regardless of your location. The field *fogDistance* is used to determine this. Low values indicate low visibility and high values indicate high visiblity. A value as high as or higher than *visibleDistance* is like have 100% visibility (unless you have *noRenderBans* un-checked).

Fog Layers

As noted above, there are three layers. Layer 1, 2 and 3. Layer1 is always the lowest and Layer 3 is always the highest. Each layer has a field *fogVolumeN* associated with it. This field takes three parameters:

Visible Distance Bottom Elevation Top Elevation

The 'visible distance' determines the distance from the camera at which visibility is (near) zero. 'Bottom' and 'Top Elevations' determine where the layer (or band) of fog begins and ends respectively. To enable a band, 'visible distance' must be greater than 0 and 'Top Elevation' must be greater than 'Bottom Elevation'. Also, do not forget, if you are going to enable more than one layer of fog, they must not overlap each other or rendering will get messed up. They may touch but not penetrate. Here is an example of some settings:

```
fogVolume0 = "250 0 50";
fogVolume1 = "350 50 150";
fogVolume1 = "25 200 500";
```


- The first layer starts at 0 meters and stops at 50 meters, with a visible distance of 250 meters.
- The second layer starts at 50 meters (touching layer one) and stops at 150 meters, with a visible distance of 350 meters.
- The third layer starts at 200 meters and stops at 500 meters, with a visible distance of only 25 meters.

Fog Color

Unfortunately, Torque only supports one fog color at this time. It looks as if it should support a general fog color and different colors for each of the layers, but the *fogVolumeColor* parameters have no effect right now. All fog has the same color: *fogColor*. *fogColor* is a specifed as a four-element vector <R G B A>, but only the first three elements have any affect. The Alpha/Intensity channel is ignored.

Visibility

OK, so we've seen that fog can affect our visibility, but how do we determine our maximum view distance? This question is critical and can affect performance as well as aesthetics. *visibleDistance* is the parameter we are looking for. It measures in meters and can be set to just about any value. A word of caution though. Extremely large distances can kill performance big time.

Wind

We have already looked at the *windVelocity* parameter, but is there anything else we should know about wind? I think, Yes, but I'll have to verify that and update this guide when I know more. – EFM

Rendering Issues

If you are having rendering issues, you may wish to check the following:

1. Get the latest drivers for your video card.
2. Set quality settings to their highest values for D3D or OpenGl, depending which you are using.
3. Be sure that BitDepth is 32 (both in your driver settings, and under Options->Graphics->Bit Depth (from SDK main menu).

If you still encouter issues, talk to some one on an IRC or post a descriptive thread (after searching the forums of course).

Sun (Mission Lighting)

The Sun object has a simple job, namely to "determine how the mission will be lit." Initially, you may or may not find this particular mission object simple to use, but with a little help this should be no big deal. Please note, that this object does not have a visible representation. That is, you can't actually see the Sun mission object. If you need a visual representation of your sun(s), use the fxSunlight mission object.

Sun Features

Some of the sun features that Torque supports are:

- Configurable Light 'Source' – Using the Sky mission object, you may configure the position of the light source and coloration (both direct and ambient) of the light it emits.
- Object Shading – Objects are darker on the side opposite the sun's position.
- Shadows – Shadows are supported, but there are issues. See 'Shadows and Sun Direction' below.
- No Sun and Multiple Suns – You can have 0, 1, 2 … well, you get the idea.

Shadows and Sun Direction

Torque supports shadows and pseudo self-shadowing. When I say pseudo self-shadowing, I mean that objects are darker on the side facing away from the sun. This is done correctly for the Terrain, Shapes, and Interiors. Unfortunately, shadows cast by objects onto other objects are a little buggy. Both Terrain and Interiors properly cast shadows onto other objects, but Shapes do not. But, what do I mean by properly? Well, shadows should be calculated based on the *azimuth* and *elevation* parameters. If I say a shadow is cast correctly, I mean it adjusts based on these parameters. This table should clarify things:

Mission Object	Shadows?	Self-Shadows?
Terrain	• Does adjust based on sun parameters. • Does affect Other Mission Objects. • Self-shadowing is baked.	Yes
Interiors (.dif)	• Does adjust based on sun parameters.	Yes

	• Does affect Other Mission Objects. • Are baked into terrain.	
Shapes (.dts)	• Does not adjust based on sun parameters. • Does affect Other Mission Objects. • Are Dynamic.	Yes

Baked shadows are calculated once during the lighting phase of a mission load and are static until/unless the mission is re-lit.

Azimuth and Altitude

Once you grasp the concept of Azimuth and Altitude, they are quite easy to work with, but describing them directly is a bit of a chore. I'm sure there is a succinct mathematical way of describing these terms, but not being a mathematician, and wanting to be clear to those similarly handicapped I will instead describe them simplistically.

Imagine if you will, we have a magic arrow (yes, a vector). The base of this arrow is stuck to the world axis. Magically, the head of the arrow always points at the sun. Given this, our magic arrow will behave as follows:

Azimuth (degrees)	Altitude (degrees)	The Arrow
0	0	Points down the Y axis and lies in the X-Y plane.
45	0	Makes a 45 degree angle between X and Y and lies in the X-Y plane.
90	45	Points down the X axis, making a 45 degree angle between X and Z.

Note: In all cases above, X, Y, and Z are the world Axes.

If that doesn't do it for you, take a look at these images, paying particular attention to the shadows cast by the hill and the small house:

Given that you are beginning to grasp azimuth and altitude, you may be wondering what the legal values are for them. Well, both can theoretically take any value between 0 and 360, but in practice, there are certain values that don't work well.

- Azimuth
  - Legal Range: [0, 360)
  - At 90 and 180 degrees shadows stop rendering.
- Altitude
  - Legal Range: [0, 360)
  - Suggested Range: [0, 90)

- Engine will crash if this is set to 90 degrees.
- Values greater than 180 are below the terrain and may produce odd effects.

Color and Ambient

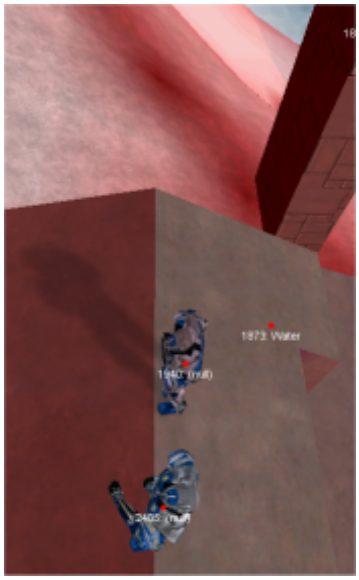
OK, enough about where the sun is, but what about the *color* and *ambient* parameters? First, both of these parameters affect the scene lighting in different ways. Briefly, *color*, is the part of the light that is cast directly onto shapes, interiors, and the terrain. It accounts for the shadows that interiors and terrain features cast. a*mbient* is the portion of the light that is scattered by the environment and appears to come from all directions. Both parameters account for the total lighting of the terrain, the character, and interiors. Shapes however, seem to ignore the *ambient* parameter. It is my opinion that the *ambient* portion of lighting is a stronger contributor than the *color* (diffuse) portion.

Both parameters take four arguments: <R G B I>, where I is the intensity. Currently, intensity has no effect for either parameter. Here are a few screen shots that demonstrate the differences between the two lighting parameters (bottom player is a static-shape):

color == 0 0 0 0
ambient == 1 1 1 0

color == 1 1 1 0
ambient == 0 0 0 0

color == 0.5 0 0 0
ambient == 0.5 0.5 0.5 0



color == 1 1 1 0
ambient == 0.5 0 0 0

Multiple Suns?

The curious in the audience may wonder, "Can I have more than one Sun?" Yes, you may, but be aware that the following is true:

- Mission Lighting will take significantly longer.
- Lighting is cumulative and clamped, meaning you can saturate your lighting.
- Shadows do not behave as you would expect with two or more light sources, instead, you'll like end up mauling your shadows.

The number one reason for adding multiple light sources is to get cool shadowing effects. Since this doesn't really work as expected, you are probably better off just sticking with one Sun.

No Sun?

This has been an on-again, off-again feature. Currently, you must specify a sun or your game will crash. However, if you want a totally dark mission, you can achieve this with a sun present. Just set the two color parameters (*color* and *ambient*) to "0 0 0 0". In the end this is safer than removing the sun, even if it does work for you 'now'.

Precipitation & Lightning

A couple of nice effects to be able to add at will are precipitation (i.e. rain, snow, hail, etc.) and lightning. These are actually separate mission objects, but I'll address them together, because they are relatively small and have at least a tangential relationship.

Precipitation Features

Some of the precipitation features that Torque supports are:

- Variable Density – You can choose between a light shower and a downpour. Additionally, the density of 'rainfall' varies randomly over time to give it a more organic feel.

- Variable Velocity – Since real 'raindrops' do not all fall at the same rate, Torque supports the ability to randomly vary the velocity of individual drops.

- Drop Coloration – For an additional degree of realism, you can modify the coloration of individual drop, by providing up to three colors.

- Multiple Textures – Because having just one texture for the 'drop' would be boring, Torque supports 16-plus. That is, you can specify just 16, or if you wish to load more than one texture file, you can specify any multiple of 16.

- Variable Drop Sizing – Although it isn't an 'out of the box' feature, you can vary the resolution and therefore the size of your drops to make them either larger or smaller.

Lightning Features

Some of the lighting features that Torque supports are:

- Target-able Strikes – You can, to some degree, target where lightning begins and where it will strike.
- Fade Color – You can choose what fade color is used for the bolts. The fade color is used to simulate the effect of seeing a lighting strike.
- Fogging – You can enable fogging features to make the lightning extra impressive, but this feature requires hardware support
- Thunder – Finally, you can supply a sound datablock to provide thunder with the lighting.

Let There Be Rain

Originally, when I looked at the precipitation object I was a bit puzzled. It had some parameters that were, well, unfathomable. Without looking at the code, and even after looking at the code, I was a bit at a loss. Fortunately, Big Dog Desmond Fletcher led the way and put together a base set of values and some sample datablocks which can be used for those parameters which I still do not understand. The short of it is that I will describe the parameters that can be modified from the inspector, but till I become more learned, I will limit my discussion of the datablocks. Instead, I will include a default set of datablocks, documented enough such that you can modify them for your use.

Fortunately, the precipitation parameters are fairly self-explanatory. We can select a previously loaded datablock (EFM see below) by selecting it from the *dataBlock* pull-down. Then, we can play with a few parameters to get the effect we want.

Precipitation Density

Precipitation density is a measure of how many raindrops we have in a certain area. We can vary the precipitation density by varying *maxRadius*, *maxNumDrop*, and *percentage*. Together, *maxNumDrops * percentage* determines the current number of 'drops' falling. We can spread these drops out by selecting various values for *maxRadius*. A low value of say 30 will cause drops to fall within 30 meters of the camera, and a value 125 will cause them to all as far away as 125 meters.

A word of caution is in order. At any one time, you will have a maximum of *maxNumDrops * percentage* drops falling. Furthermore, the absolute maximum is capped at 2000 drops. So, if the *maxRadius* is something small and the camera is moving at high speeds, the camera could 'punch through' the perimeter, which would look kind of weird.

Precipitation Velocity

In order for our precipitation to look more 'realistic', we'll want it to fall at varying rates. To do this, simple set *minVelocity* to a non-zero value lower than *maxVelocity*. Now, drops will fall at some random speed between *minVelocity* and *maxVelocity*. Additionally, setting *offsetSpeed* to a non-zero value adds a bit of horizontal velocity to the drops. Don't overdo it on this parameter though as high values can make the precipitation look a bit unnatural.

Varying Drop Colors

The base color of your drops is determined by the texture(s) you use for your precipitation (see below), but you can modify this with the *color[3:1]* parameters. As far as I can tell, 33% of the drops are either *color1*, *color2*, or *color3*. So, setting the <r g b> portion of these to something other than <1 1 1> will cause the textures to be shaded that color. Note, the alpha channel (fourth value) does nothing.

Precipitation Media

By default, any individual 'drop' is a billboard. These billboards are textured with a sub-texture from a PNG file specified in your DML file. Sub-textures are arranged in a 4-by-4 matrix of images (see sample image below). You specify the DML file in your precipitation datablock in the *materialList* parameter.

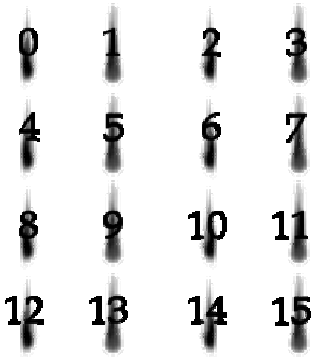


Fig X. Precipitation Texture
(a 4 x 4 grid of subtextures)

If you need more than 16 variations on your 'drops', you can specify additional textures in your DML file. If you haven't read the section above on they Sky object, a DML file is a media file of the format:

```
~/relative_path/media_name <-- Texture 0
~/relative_path/media_name <-- Texture 1
...
~/relative_path/media_name <-- Texture 7
```

The path is relative to the game's starting directory (i.e. where the executable lives). The media_name is the name of a texture without its extension. You may use up to eight JPG or PNG files for precipitation, but I suggest using PNG as JPG does not support transparency which you will likely need. Oh, and if you are only using one texture, be sure the DML file ends with a blank line (i.e. add a carriage return to the final line in the file or else you'll just get grey-blocks for drops).

'It was a dark and stormy night…'

What would a storm be without a little lightning and thunder? Well, fortunately you don't need to find out, because Torque comes with a Lightning mission object.

Lightning objects are blocks like water. This means, that you can place multiple blocks of lightning throughout your mission, or if you choose, you can have just one really big block covering the whole mission. Blocks may overlap. You can freely scale the lightning block using the inspector the mouse.

Lightning Strikes!

First, it is important to understand what a strike is. When engine gets ready to draw the lightning, it decides whether it is going to strike the ground, the highest local object, or if there will be a miss.

When there is a miss, the lightning is drawn at an angle, sometimes even parallel to the ground. These misses, give the lightning a more realistic look. So, how does the engine determine if there will be a miss or a strike, and when there is a strike, how is it determined if an object will be hit, or the ground?

First, the zone where anything can be hit is determined by the location of the lightning box as well as the *strikeRadius*. Bolts will strike objects, or the ground within *strikeRadius* of the lightning object. To determine if an object will be hit, or if the ground will be struck, then engine grabs a list of all damageable objects in the 'strike zone' and does a sort, looking for the highest object. It can randomly choose an object that is not the highest, but it has a preference for the highest object (as does real lightning). Finally, the engine rolls the dice so to speak and if the value it gets back is less than or equal to *chanceToHit* (remember those good old AD&D days?), whamo that object gets hit. If the value is higher than *chanceToHit*, then the bolt hits a random location on the ground.

We can control the number of lighting strikes (this includes misses) per minute with the parameter *strikesPerMin*. This is not the inverse of the strike period, but instead a rough number of strikes per minute. Increasing this value increases the number of strikes in any period of time, but strikes can happen very rapidly or with short pauses between them. This just gives it a more random feel. You can't predict a lightning strike…

So, what about *strikeWidth*? Well, this determines the width of the bolt on a strike. Bolts all have a default width for misses, but for strikes you can control the width. Do you want a really fat strike or a really narrow one?

Lightning Color

The textures you choose for your lightning are used as a mask, but the coloration comes from the *color* and *fadeColor* parameters. The bolts are drawn first, using *color* and then over a short period of time they are faded out. While this fade occurs, the bolts are colored *fadeColor*. This gives a nice heated plasma effect and mimics the behavior of the eye when it sees a lightning bolt. When you see an actual lightning strike or any focused bright light, most of the receptors in the eye-ball fire for the area where the bolt is focused by your eye's lens. This temporarily uses up all the available chemicals which

make sight possible. In other words, those receptors are temporarily turned off by the 'overload'. The effect is a phantom bolt which fades over a short time.

Leaning Lightning?

In addition to controlling the strike zone, we can control where the lighting bolts start from. If we set *boltRadius* to zero, then all bolts will radiate from the topmost center position of the lighting box. Alternately, we can set the value to something really big, like 500. Now, all the bolts will see be be coming from far away and angling towards the strike zone (assuming a small strike zone of course).

Ooooh Pretty Lightning

Finally, if you set *useFog* to true and if the user's graphics card supports both multi-texturing and fog coordinates extensions (a pretty good bet cards two or fewer years old), the engine will do a nice bit of texturing with local fog (i.e. fog around the camera).

Lightning Datablock

You must predefine a datablock before you can place a lightning object. However, there isn't much to this. There are only two parameters to specify in the datablock.

Field Name	Description
strikeTextures[8]	Eight texture slots for relative paths and names of lightning texture DML files. **Only use slot 0.**
thunderSounds[8]	Eight sound profile slots for thunder/lightning strike sounds.

Now, it may seem a little odd that there are eight texture slots when we are using a DML file, and in point of fact, it is. I think this is a bit of legacy code that didn't get straightened out. For now, just use the first slot to specify the location of your DML file. in your DML file, you may specify the paths and names of up to eight lightning textures. These textures should have a black background. The engine will use that to mean the area is to be transparent. Non-black areas are translucent.

Where is the Sound?

I have deferred discussing sound because, frankly, I haven't gotten it to work consistently yet. Until sound is worked out, you can use a 2D audio emitter (see audio emitters below). I promise, this section will get updated as soon as I understand this aspect of both Precipitation and Lightning objects.

Audio Emitters

So far, we've focused on visible environmental objects. What about sounds? Audio Emitters are an object that you can use for placing positional sounds. Audio emitters have the ability to turn themselves on and off based on a trigger. This trigger can be modified in size and shape to meet your needs. Let' take a look, or perhaps I should say, let's have a listen?

Audio Emitter Features

Some of the audio emitter features that Torque supports are:

- 2D Sound – This is sound with no apparent source. In other words it is neither directional nor positional.
- 3D Sound – This is sound with a specific source. Furthermore, this type of sound is modulated by distance from and facing angle to the sound source.
- Looping and Non-Looping Sounds – Emitters can be programmed to loop a variable number of times or as one-shot emitters.
- Triggers – 3D sound emitters have the ability to turn themselves on and off based on a cut-off distance.

2D Sound

2D sound is very simple. All 2D sound emitters are turned on at the earliest opportunity (which is some time during the game load process). If looping is enabled (see below), audio emitters will not stop playing until all loops have been exhausted, otherwise they will play once and then stop.

You can specifying a 2D audio emitter as follows:

- profile
  - *profile* – <NULL>
  - *useProfileDescription* – unchecked
- Media
  - *description* – Relative directory+filename for the sound file
    - Only .WAV format is supported.
    - Mono and Stereo Formats OK
    - Example in inspector: fps/data/sound/testing.wav
    - Example in mission file: ~/data/sound/testing.wav
- Media
  - *type* – 1..31 (see '2D Gain' below)
- Sound
  - *volume* – Between 0.0 (0% gain) and 1.0 (100% gain)
  - *outsideAmbient* – checked
- Set looping parameters (see 'Looping' below)
- Advanced

- *is3D* – unchecked.

2D Gain

Gain determines how loudly your sound will play. The gain equation for 2D emitters is as follow:

*2D Gain == 'Game Master Volume' * 'Audio Group Gain' * 'Emitter Gain'*

'*Game Master Volume*' – is controlled from the SDK front panel under Options->Audio.

'*Audio Group Gain*' – is controlloed by the field Media->*type*.

- Valid values for type are 1..31. By default, only 1..8 are set up.
  - 0 – Is reserved
  - 1 – GUI Audio Type (Options->Audio->Shell Volume)
  - 2 – Sim Audio Type (Options->Audio->Sim Volume)
  - 3..8 – Set to 0.8. (Search for 'channelVolume' in scripts).
- The purpose of this gain is to allow you to adjust the gain for a group of emitters in one step.

'*Emitter Gain*' – is controlled by the field Sound->*volume* parameter.

Looping

If you haven't already guessed, the looping parameters allow you make an emitter (2D or 3D) play the sound file between 1 and infinite times. To enable looping make sure Looping->*isLooping* is checked. Then, set your loop count. Loop counts work as follows:

- *loopCount* == -1 – Loop infinitely.
- *loopCount* == 0 – Loop once and only once.
- *loopCount* == 1 – Loop once, possibly twice.
- *loopCount* == (N > 1) – Loop N times.

On rare occasions, a value of 1 will cause <u>two</u> loops. So, if you really want only one loop, use a loopCount setting of 0.

Loop Gaps

The Loop Gap parameters control the delay between subsequent loops. *minLoopGap* as you would imagine defines the lower boundary for delays, and *maxLoopGap* the upper. Torque randomly chooses a value between these two. Loop Gaps are approximately equal to 2 * N milliseconds, where N is the LoopGap value selected. Please note that Loop Gaps can be used to do some interesting things:

minLoopGap	maxLoopGap	Action
0	0	Sound turns on, but won't turn off (2D and 3D)
0	1	Sound turns on immediately and turns off at end of loop or upon exitting 3D region (see below).
1	0	Sound does not turn on, *ever*.
N > 1	N > 1	Normal behavior.

By using the settings *minLoopGap* = 1, *maxLoopGap* = 0 you can tell the emitter to not play at load time. Once the load is completed, you can have a script set the Gap values to whatever delay you need or you can hook the sound up to a trigger…

2D Visual Feedback

Visual feedback in 2D mode is pretty simple. While editing, you can see the emitter as a small cube. The cube will be black while not playing and green while playing.

NOT PLAYING IMAGE
PLAYING IMAGE

3D Sound

In real life, sound radiates from a source to a listener. Additionally, sound is attenuated by several factors, including distance, angle, occlusion, etc. Torque simulates the behaviour of real-world sound OpenAl's 3D sound features. 3D audio emitters support distance and angular attenuation. How they support these features can be a little confusing so we'll approach the topic piecewise with a roll-up at the end.

Sound Zones and Sound Cones

In practice, audio emitters support four zones of sound:

Zone	Description	Gain Attenuation
A	Listener in Inner Cone	Gain is a function of linear distance from source.
B	Listener in Outer Cone	Gain is a function of linear and from source and angular distance from Inner Cone edge.
C	Listener in area outside Outer Cone.	Gain is a constant value determined by Outside Volume.
D	Listener beyond maximum distance from source.	Near zero gain. Emitter is deactivated (eventually).

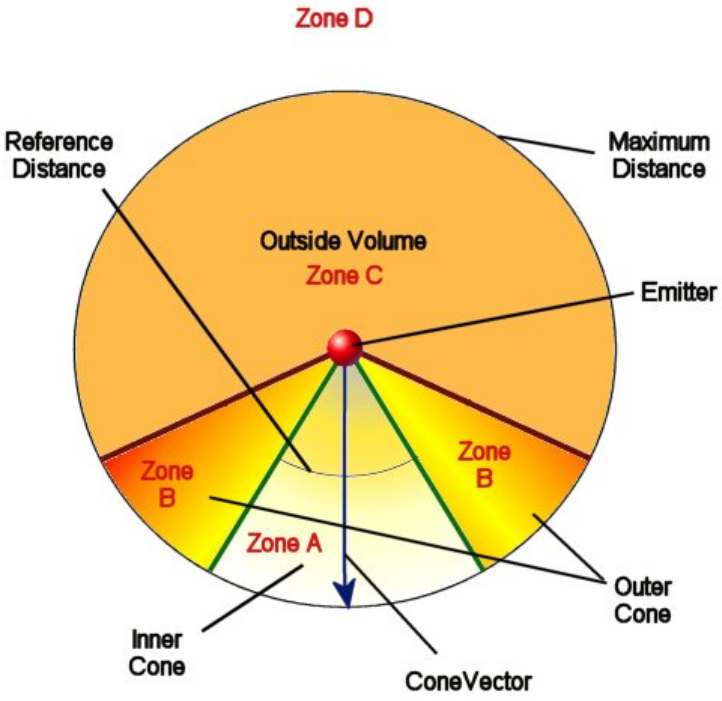


Figure X. Sound Cones

Zone A – Inner Cone

As noted above, gain in the inner cone is a function of distance from the emitter (source). To determine the physical volume of the inner cone, we must specify the following:

- *is3D* must be checked to enable 3D sound.
- *position* specifies the tip of the cone and the base of the *coneVector*.
- *rotation* specifies the direction in which *coneVector* points.
- *maxDistance* specifies base of the cone. *coneVector* is a unit vector, but you can image a line passing through the vector, starting at *position* and ending at *position* + *coneVector* * *maxDistance* this is the position of the cone base.
- *coneInsideAngle* specifies the inner cone sweep.

To specify the gain of the inner cone, we must specify the following:

- *volume* – Emitter gain.
- *referenceDistance* – This specifies the distance (from the emitter) at which 3D gain == 0.5.

Inner Cone gain works as follows,

Listener Position	Emitter Gain
P < R	0.5 * P/R
P == R	0.5
M > P > R	~ R/P

where,

P = | listener position – emitter Position |
R = *referenceDistance*
M = *maxDistance*

Zone B – Outer Cone

Gain in the outer cone is a function of inner cone gain and the angle from the outer edge of the inner cone. To determine the physical volume of the outer cone, we must specify the following:

- Inner Cone

- *coneOutsideAngle* specifies the outer cone sweep.

The outer cone shares all the parameters of the Inner Cone including the axis. To specify the gain of the inner cone, we must specify one additional parameter:

- *coneOutsideVolume* – Gain at and beyond outer edge of outer cone.
    - **Imporatnt!** If this value is 0, the outer cone will be disabled and there will be no sound except inside the inner cone.

Outer Cone gain works as follows,

Listener Position	Emitter Gain
Ca == Ia	Ig
Ca < Ia < Oa	Ig -> *Ov* (as a function of angle)
Ca == Oa	Ov

where,

Ig = Inner cone gain at current distance from emitter.
Ca = (*coneOutsideAngle* – Current Angle) / 2
Ia = *coneInsideAngle* / 2
Oa = *coneOutsideAngle* / 2
Ov = *coneOutsideVolume*

Zone C – Outside Volume

If *coneOutsideVolume* is non-zero, the area outside of Outer Cone has a gain between *coneOutsideVolume* / Distance from emitter.

Outer Volume (zone) gain works as follows,

Listener Position	Emitter Gain
P	*coneOutsideVolume -> 0* (as a function of distance)

P = | listener position – emitter Position |

Zone D – Beyond *maxDistance*

The *maximumDistance* can be used to draw a imaginary sphere around the emitter. If the camera enters that sphere, the emitter is told to load its sound. Additionally, if the camera is inside an enabled sound zone, the emitter is told to play the sound. Conversely, if the camera moves from within the sphere to outside the sphere, the sound is told to stop playing. This doesn't mean the sound will stop immediately however. There will be some (variable) delay. This means, that gain will be further attenuted outside the sphere, as a function of distance, until the sound is no longer audible (if the camera is far enough away) or until the sound eventually stops playing on its own.

3D Visual Feedback

Before we jump into examples, let's discuss the visual feedback associated with 3D Audio Emitters. Because, there are more audio concepts to express, the visual feedback is a little more complex than for 2D emitters, but only marginally.

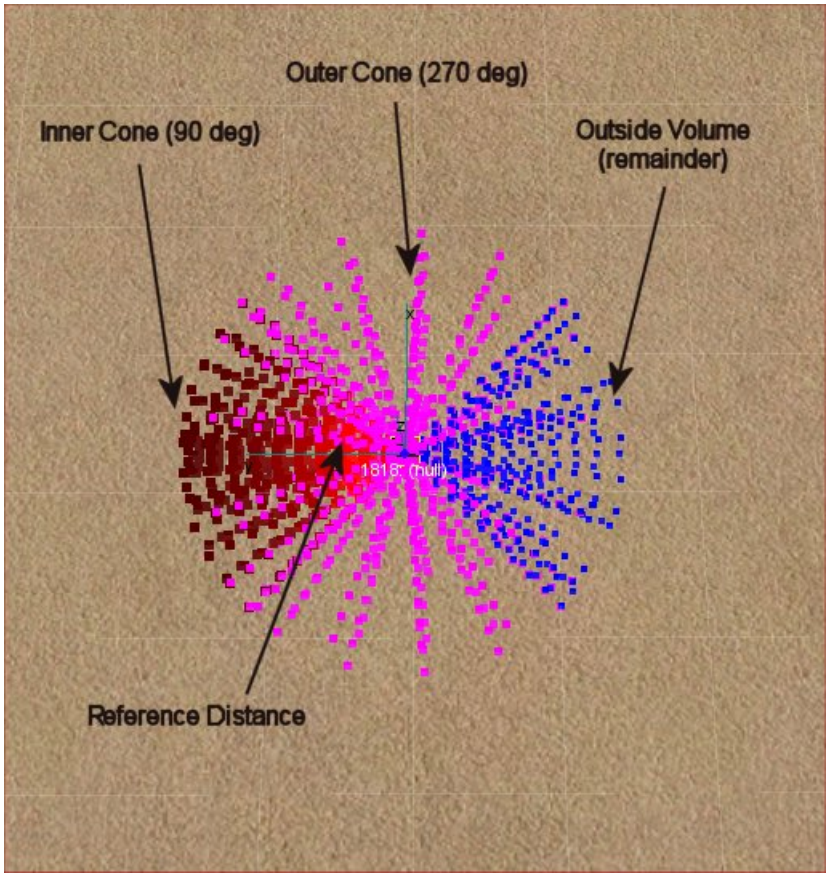


Figure X. Audio Emitter – 3D Visual Feedback

Inner Cone	Red fading to black. Fade starts at *referenceDistance*.
Outer Cone	Pink (Purple?…your call)
Outside Volume	Blue
On/Off indicator	Same as 2D (not visible in figure X).

You can specifying a 3D audio emitter as follows:

- profile
  - *profile* – <NULL>
  - *useProfileDescription* – unchecked
- Media
  - *description* – Relative directory+filename for the sound file
    - Only .WAV format is supported.
    - Mono and Stereo Formats OK

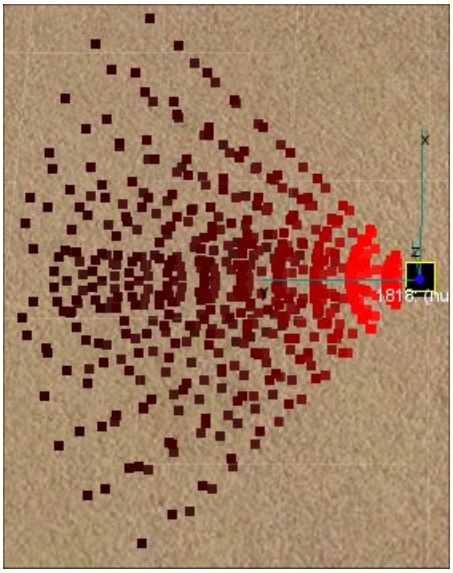
        - Example in inspector: fps/data/sound/testing.wav
        - Example in mission file: ~/data/sound/testing.wav
- Media
    - *type* – 1..31 (see '2D Gain' below)
- Sound
    - *volume* – Between 0.0 (0% gain) and 1.0 (100% gain)
    - *outsideAmbient* – checked
- Set looping parameters (see 'Looping' above)
- Advanced
    - *is3D* – is checked.
    - *coneInsideAngle* – Set to your preference.
    - *coneOutsideAngle* – Set to your preference. 0 to disable.
    - *coneOutsideVolume* – Set to your preference. 0 to disable all but Inner Cone.
    - coneVector – No. Don't touch this. It is set automatically when you adjust rotation. Changes will be over-ridden.

MONO Only!

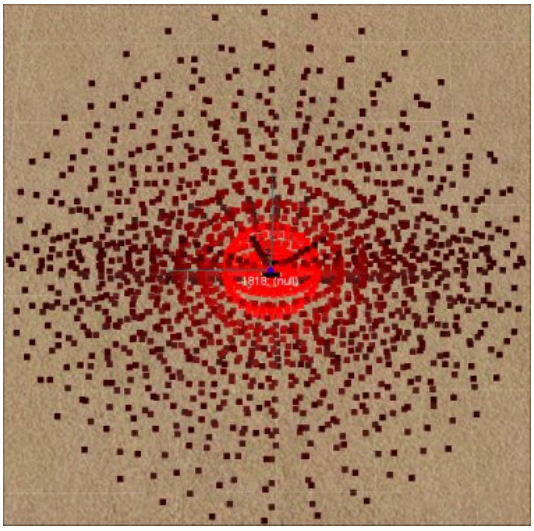
Unlike 2D audio emitters, 3D audio emitters can only use MONO sound files. If you use a STEREO audio file, the sound will not correctly attenuate. In other words, stereo files always have a gain of 1.0.

3D Emitter Examples

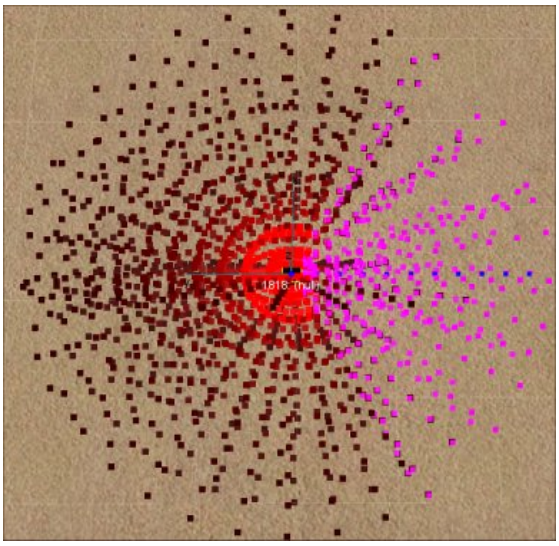
Before discussing advanced audio emitter topics, lets look at some example emitters:



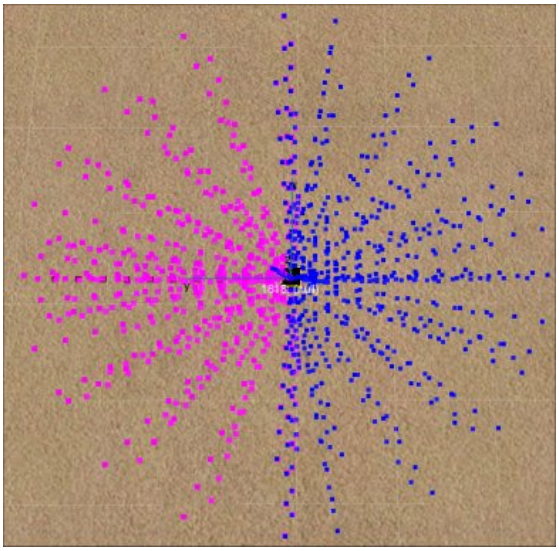
coneInnerAngle: 90
coneOuterAngle: 0
coneOutsideVolume: 0



coneInnerAngle: 360
coneOuterAngle: 0
coneOutsideVolume: 0



coneInnerAngle: 270
coneOuterAngle: 360
coneOutsideVolume: > 0



coneInnerAngle: 180
coneOuterAngle: 360
coneOutsideVolume: > 0

Profiles

EFM - INCOMPLETE

Audio Descriptions

EFM - INCOMPLETE

Caution!

EFM - INCOMPLETE

Particle Emitter Nodes

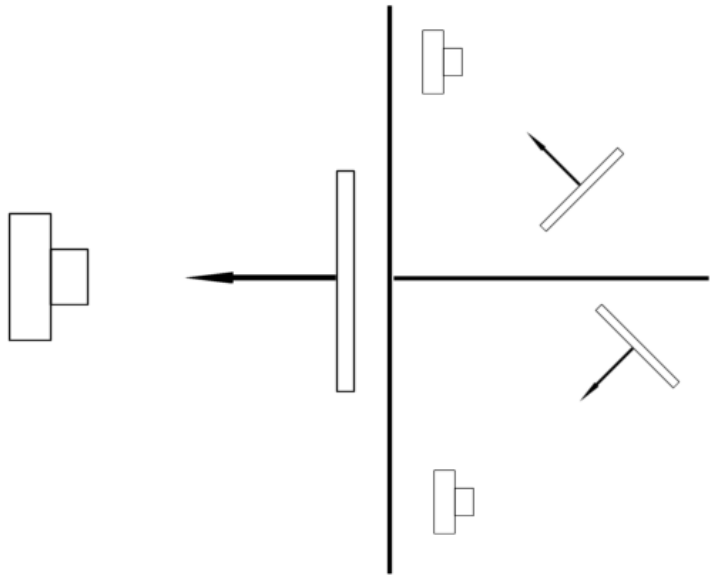
One of the more time consuming mission objects to place are Particle Emitters. No, not because they are particularly hard to understand, but because they offer a venerable cornucopia of features. And, they're just plain fun to play with! In fact, if you don't approach them knowing the basics of how to use them and with a good idea of end result you want, you could burn several hours goofing around. While I can't help you focus on a particular idea, I can help you understand the basics of using them.

I must warn you before we start, we are going to depart from using the mission editor alone. In order to build emitters, we need to write some scripts, well not scripts really, but some datablocks. A data whatzit? A datablock. I'm not going to describe the purpose of datablocks here. Nor will I cover all their syntax. For that you'll have to head to the Scripting chapter of Tech School. For now, you can just use my examples directly and you should not get into too much trouble.

What is a Particle Emitter Node?

Particle Emitters Nodes (PENs) are static objects (that is they don't normally move), which can be used to provide special effects such as: smoke, fire, waterfalls, fireflies, … you name it. They do this by emitting, you guessed it, particles. As is commonly[2] the case in 3D systems, these particles are billboards[3]. For the sake of this discussion, think of a billboard as a polygon that automatically orients itself to be facing a specific direction. In the case of particles, these billboards are usually textured with a partially opaque and partially translucent texture and are usually facing the camera. What this means is that when you look at any particular particle it will normally be facing you and you will likely be able to see through parts of it.

[2] Some common particles are billboards, pixels, and lines.
[3] If you want to learn all about billboards, pick up a good book like Moeller and Haines 'Real-Time Rendering'



Screen Oriented Billboard
(faces camera at all times)

So, what do we have so far? In Torque, particles are billboards and they get spewed out of Particle Emitter Node. Great, is that all? No. Particles don't just spew out of PENs. In fact, we (the game designers) choose how many particles there are, what kinds of visual effects they have, how fast they spew, whether they are affected by wind, gravity, etc… It is all these factors that make PENs useful. Most important of all we can create some really awesome effects at a low[4] cost.

Particle Emitter Data Blocks

Like I mentioned above, we need to build a few datablocks before we can play with particle emitters. Specifically, we will need a minimum of three databocks:

- **ParticleEmitterNodeData (PEND)** – Think of this as the base for the emitter. It controls one aspect of the particle emitter, Time.

- **ParticleEmitterData (PED)** – This is used to describe the behavior of the PEN itself. It controls how many particles are emitted, how fast, and in what position/direction.

- **ParticleData (PD)** – This describes individual particles. It controls coloration, fade, spin, drag, velocity, acceleration, whether a particle is affected by gravity, particle life… and a few other things. You'll see.

[4] Done right, particles do not consume a lot of resources (memory, CPU time, geometry budget, etc.).

You will find the following base datablocks in the EGT Lesson kit (path):

```
// EGT_ParticleEmitterDBs.cs
datablock ParticleEmitterNodeData(EGT_BasePEND)
{
   timeMultiple = 1;
};

datablock ParticleEmitterData(EGT_BasePED)
{
   ejectionPeriodMS      = 200;
   periodVarianceMS      = 100;
   ejectionVelocity      = 6.0;
   velocityVariance      = 0.0;
   ejectionOffset        = 0.0;
   thetaMin              = 0.0;
   thetaMax              = 0.0;
   phiReferenceVel       = 0.0;
   phiVariance           = 0.0;
   overrideAdvance       = false;
   orientParticles       = true;
   orientOnVelocity      = false;
   particles             = EGT_BasePD;
   lifetimeMS            = 0;
   lifetimeVarianceMS    = 0;
   useEmitterSizes       = false; // Not used for PENs
   useEmitterColors      = false; // Not used for PENs
};

datablock ParticleData(EGT_BasePD)
{
    textureName          = "~/data/shapes/particles/bluesphere";
    dragCoefficient      = 0.1;
    windCoefficient      = 0.0;
    gravityCoefficient   = 0.0;
    inheritedVelFactor   = 0.0;
    constantAcceleration = 0.0;
    lifetimeMS           = 1000;
    lifetimeVarianceMS   = 0;
    spinSpeed            = 0.0;
    spinRandomMin        = 0.0;
    spinRandomMax        = 0.0;
    useInvAlpha          = false;
//    animTexName          = false;
    animateTexture       = false;
    framesPerSec         = 1;
    colors[0]            = "1.0 0.0 0.0 0.0";
    colors[1]            = "1.0 1.0 1.0 1.0";
    colors[2]            = "0.0 0.0 1.0 1.0";
    colors[3]            = "1.0 1.0 1.0 1.0";
    sizes[0]            = 1.0;
    sizes[1]            = 1.0;
    sizes[2]            = 1.0;
    sizes[3]            = 1.0;
    times[0]            = 0.0;
    times[1]            = 0.33;
    times[2]            = 0.66;
    times[3]            = 0.1;
};
```


After specifying the above datablocks (once), we can create as many PENs as we like (in our mission file), using the following simple format:

```
new ParticleEmitterNode(PEN_Test0) {
      position = "0 0 0";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      dataBlock = "EGT_BasePEND";
      emitter = "EGT_BasePED";
      velocity = "1";
   };
...
new ParticleEmitterNode(PEN_TestN) {
      position = "10 10 0";
      rotation = "1 0 0 0";
      scale = "1 1 1";
      dataBlock = "EGT_BasePEND";
      emitter = "EGT_BasePED";
      velocity = "1";
   };
```

These may look daunting, but don't worry. They really aren't. Let's go ahead and discuss the individual parameters, miscellaneous important equations, and then try building some PENs.

ParticleEmitterNodeData (PEND) Datablock Parameters

The PEND datablock specifies a time multiplier for and individual PEN. This time is used subsequently in certain calculations, which we'll cover in the equations section.

Parameter	Range	Description
timeMultiple	[0.01, 100.0]	Time multiplier, used to increase or decrease elapsed time by a ratio. Affects ejection period, ejection position calculation.

ParticleEmitterData (PED) Datablock Parameters

The PED datablock specifies the behavior of a PEN, including what particles it emits, at what rate, in what direction, with how much velocity, and for how long. It also describes how particles will be oriented.

Parameter	Range – Default	Description
ejectionPeriodMS	[1, INF] - 100	Milliseconds between last and next particle ejection.
periodVarianceMS	(0, ejectionPeriodMS] - 0	Amount to vary ejection period by.
ejectionVelocity	[0, INF] - 2.0	Initial velocity imparted to particles.
velocityVariance	[0, ejectionVelocity] - 1.0	Amount to vary initial velocity by.
ejectionOffset	[0, INF] - 0.0	Particle ejections begins at *ejectionOffset* distance from emitter.
thetaMax	[0, 180] [thetaMin, 180] - 90.0	Modifies emitter ejection up and down. This modifies the PEN up vector. 0 = fully up, 180 = fully down
thetaMin	[0, 180] [0, thetaMax] - 0.0	Modifies emitter ejection up and down. This modifies the PEN up vector. 0 = fully up, 180 = fully down

phiReferenceVel	[0, 360]	Causes emission point to rotate clockwise N degrees per second about the PEN UP vector.
phiVariance	[0, 360]	Separate from *phiReferenceVal*, this parameters enables a random ejection between 0 degress and *phiVariance*.
overrideAdvance	false	Always false (legacy code).
orientParticles	true or false	true – Face emission direction. false – Face camera.
orientOnVelocity	true or false	true – If *orientParticles* == true, face direction of motion. false – Use *orientParticles* setting.
particles	PD name(s)	List of PD datablocks to use/emit.
lifetimeMS	[0, TBD]	Length of time to eject particles before stopping (in milliseconds). N == 0 – Always on N == >0 – N Milliseconds
lifetimeVariance	[0, lifetimeMS)	Amount to vary lifetimeMS by.
useEmitterSizes	false	Not used for PENs. These apply to particle emitters attached to a particle emitter object (See Particles chapter of Tech School).
useEmitter Colors	false	Not used for PENs. These apply to particle emitters attached to a particle emitter object (See Particles chapter of Tech School).

ParticleData (PD) Datablock Parameters

The PD datablock describes an individual particle, including how it is a affected by things like wind, drag, gravity, and an acceleration factor. It also desribes physical parameters of the particle including color, size, spin, and lifetime. Lastly, it describes advanced features, like alpha inversion, and animation.

Parameter	Range - Default	Description
dragCoefficient	(0, TBD] - 0.0	Factor determining velocity subtracted per second.
windCoefficient	[0, 1.0] - 1.0	Percentage of wind vector added to particle vector.
gravityCoefficient	[TBD, TBD] - 0.0	Gravitational acceleration for particle. Negative values cause particles to rise.

inheritedVelFactor	[TBD, TBD] - 0.0	Multiplier determining how much of the PED. ejectionVelocity is added to the initial velocity of the particle.
constantAcceleration	[TBD, TBD] - 0.0	Incremental velocity added to particle velocity on a per-second basis.
lifetimeMS	[100, TBD] - 1000.0	Particle life in milliseconds. At the end of its life, the particle is deleted.
lifetimeVarianceMS	[100, lifetimeMS] - 0.0	Amount to vary lifetimeMS by.
spinSpeed	[-10000, 10000] - 0.0	Speed at which particle rotates about its facing vector. Only valid when PED.*orientParticles* == false
spinRandomMin	[-10000, 10000] - 0.0	Minimum random value added to *spinSpeed*.
spinRandomMax	[-10000, 10000] - 0.0	Maximum random value added to *spinSpeed*.
useInvAlpha	true or false - false	Inverts interpretation of texture alpha.
animateTexture	true or false - false	Sequence between additional textures, specified in *animTexName[50]*.
framesPerSec	[1, 200] - 1	Frame frequency for animated textures.
textureName	“Path + File Name” NULL	Texture path and filename (PNG only). Must be <= 255 characters long
animTexName[50]	“Path + File Name” NULL	Additional texture path and filenames (PNG only). Used when *animateTexture* == true. *animTexName[0]* same as *textureName*
colors [4]	<R, G, B, I>	Color interpolation values. Note: Only these values determine particle color. The texture is used as an alpha-map, not for coloration.
sizes[4]	[0, TBD]	Size interpolation values.
times[4]	[0, 1]	Key frames. These affect interpolation rates over life of particle.

PEN Parameters

In order to specify a PEN in your mission, you can add it with the Mission Editor (ME) (F11-> F4; Mission Objects -> environment -> particleEmitter), or by hand editting your mission file. In order to do this, we need to specify the following parameters.

Group	Field Name	Description
Transform	*position*	Used to set location of PEN
	rotation	Values have no effect
	scale	Values have no effect
Misc	*nameTag*	TBD
	dataBlock	PEND datablock name
	emitter (Particle data in ME)	PED datablock name
	velocity	Initial ejection velocity for this emitter

PEN Equations

As promised, I'll describe some important equations below. Armed with these and the subsequent descriptions of Theta & Phi, Orientation, and Animation, you should be able to pre-specify approximate values before you start to experiment and tune, which should save lots of time.

Note: Some of the equations below produce vectors. These vectors are calculated from a series of vectors and scalars (from the datablocks and internally from the engine). In order to be clear which is which I will underline vectors and **bold scalars** in the equations below. Furthermore, blue variables are from datablocks and will have the datablock prefixes, wheras red variables are from the engine. Lastly, unless otherwise specified input vectors are unit vectors. Oh, one more thing, velocities are in meters per second.

Particle Initial Velocity

Each particle is given an initial velocity vector at ejection time. The velocity vector is determined as follows:

```
emitAxis x PEN.velocity x ejectionAxis x ( PED.ejectionVelocity +
PED.velocityVariance * 2.0 * rand[0.0,1.0] - PED.velocityVariance )
```

```
emitAxis is always <0, 0, 1> (in theory you can ignore this factor)

ejectionAxis depends on orientation, theta, and phi

rand[0.0,1.0] == Random value between 0 and 1.0.
```

Particle Post-Ejection Velocity Changes

After being ejected, a particle may or may not have its velocity modified.

```
NextVelocity == CurrentVelocity *
     ( (PD.constantAcceleration * InitialVelocity) -
       (CurrentVelocity * PD.dragCoefficient) -
       (WindVelocity * PD.windCoefficient) +
       (<0,0, -9.81> * PD.gravityCoefficient) )

Note: There is a time delta component not shown.
```

Particle Lifetime

Particle lifetimes are a simple concept. If a particle is created at time N, at time N + lifetime, the particle will be deleted. Lifetimes affect interpolation, which will describe next. The PD.lifetimeVarianceMS allows us to randomly vary individual lifetimes which makes things seem less artificial when viewed. Lifetimes are of course in milliseconds.

```
PD.lifetimeMS + ( rand[-1,1] x PD.lifetimeVarianceMS )
```


Particle Interpolations

Particles are subject to two types of interpolation, color and size. Color interpolation is the ability to modify the particle color over its lifetime. Similarly, size interpolation is the ability to modify the particle size over its lifetime.

Interpolation is controlled by keyframes (*PD.times[4]*), of which Torque allows up to four. The minimum value for a key frame is 0.0 and the maximum value is 1.0. Keyframes should be used in order and unused key frames should be set to 1.0.

This is probably all sounding rather mysterious still so I'll give some examples and explain what they do.

```
PD.color[0] = "1.0 1.0 1.0 1.0";
PD.color[1] = "1.0 1.0 1.0 0.0";
PD.color[2] = "1.0 1.0 1.0 0.0";
PD.color[3] = "1.0 1.0 1.0 0.0";

PD.size[0] = 1.0;
PD.size[1] = 1.0;
PD.size[2] = 1.0;
PD.size[3] = 1.0;

PD.time[0] = 0.0;
PD.time[1] = 1.0;
PD.time[2] = 1.0; // Unused
PD.time[2] = 1.0; // Unused
```

The above example tells the particle to remain at size 1.0 for its entire lifetime and to fade smoothly from Bright White to transparent.

```
PD.color[0] = "1.0 0.2 0.2 1.0";
PD.color[1] = "0.2 1.0 0.2 1.0";
PD.color[2] = "0.0 0.2 1.0 1.0";
PD.color[3] = "0.0 0.2 1.0 1.0";

PD.size[0] = 0.5;
PD.size[1] = 1.0;
PD.size[2] = 1.5;
PD.size[3] = 2.0;

PD.time[0] = 0.0;
// 1/3 time here framed by time[0] and time[1]
PD.time[1] = 0.33;
// 1/3 time here framed by time[1] and time[2]
PD.time[2] = 0.66;
// 1/3 time here framed by time[2] and time[3]
PD.time[2] = 1.0;
```

The above example causes the particle to smoothly increase from a size of 0.5 to 2.0 over the particle's lifetime. Additionally, the particles color is interpolated from a shade of red, to green, then to blue, where it stays for the last 1/3 of its lifetime.

Interpolation takes some practice getting used to, but it's a nice touch which gives us some cool variations on particles.

PEN Lifetimes

As particles have lifetimes, so can particle emitter nodes. A PEN can be told to emitt particles forever or for a fixed duration.

```
// Emit forever after being created
PED.lifetimeMS = 0;
```

```
// Emit for five seconds plus or minus 1.5 seconds after being created
PED.lifetimeMS = 5000;
PED.lifetimeVarianceMS = 1500;
```

PEN Particle Ejection Frequency

The PEND and PED datablocks give us parameters to adjust the rate at which particles are emitted. *PEND.timeMultiple* acts as a multiplier for the *PED.periodMS* and *PED.periodVarianceMS* parameters.

```
// Emit a new particle every 200 miliseconds with no variation
PEND.timeMultiple    = 1.0;

PED.periodMS         = 200;
PED.periodVarianceMS = 0.0;
```

In the above example time is advanced in a 1 to 1 ratio so the PED parameters control frequency.

```
// Emit a new particle every 100 miliseconds with no variation
PEND.timeMultiple    = 0.5;

PED.periodMS         = 200;
PED.periodVarianceMS = 0.0;
```

In the above example time is advanced in a 1 to 2 (simulation seconds versus real seconds) ratio so the PED parameters are in practice halved.

```
// Emit a new particle every 400 miliseconds +/- 100 ms
PEND.timeMultiple    = 2.0;

PED.periodMS         = 200;
PED.periodVarianceMS = 50;
```

In the above example time is advanced in a 2 to 1 (simulation seconds versus real seconds), and the local period variance is 50 ms. This really equates to emitting a particle every 300 thru 500 ms.

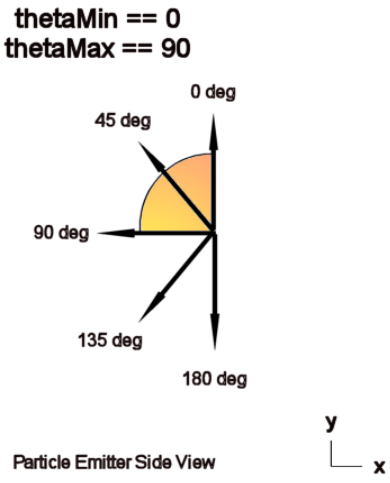
Theta and Phi Explained

Remember those PED parameters Theta and Phi something..something? They control the orientation of the emitter. In other words, by varying these parameters, you can point the emitter more up or down and more left or right. Let's talk about theta first since it is the simpler of the two.

Theta

Theta controls the up and down of the emitter's ejection vector. Imagine, if you will, that you are standing to the side of an emitter. If we play with the Theta parameters, we can make the emitter eject particles anywhere straight up and straight down.

Torque supplies the two parameters PED.*thetaMin* and PED.*thetaMax*. These act as boundaries. We point the emitter in a specific direction such as 90 degrees (straight out) by merely setting PED.*thetaMin* to 90 and PED.*thetaMax* to 90. Alternately, if we wish to spread our particles out, we can set PED.*thetaMin* to 0 and PED.*thetaMax* to 90. Now, particles will be randomly ejected with an ejection vector pointing between straight up and straight out respectively.



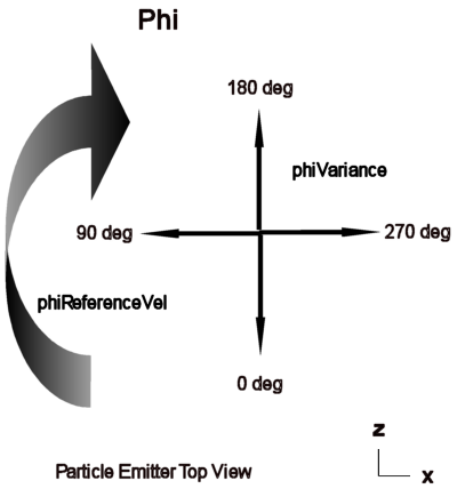
Phi

Since theta was so simple, you might jump to the conclusion that Phi controls the left and right. If you did, you would be both right and wrong.

Right. The phi parameters do control the ejection vector's left to right pointing, but not like the theta parameters.

Wrong. Whereas PED.*thetaMin* and PED.*thetaMax* were used to set the minimum and maximum up-down ejection angles. Our minimum Phi angle is always zero degrees and PED.*phiVariance* controls the upper angle.

This means we cannot point our phi in the same way we can theta. I know, bummer! Personally, I believe this needs to be changed or added to[5], but for now that is what you get. So, what about PED.*phiReferenceVel*? This strange parameter causes the emitter to spin clockwise about its UP vector. As it spins, the relative pointing direction of 0 degrees changes. PED.*phiReferenceVel* is measured in degrees per second.



OK, let's summarize what the Theta and Phi parameters do for us. PED.*theataMin* and PED.*thetaMax* allow us to control the up-down pointing of our ejection vector. Furthermore we can specify a range of up-down positions between which the ejection vector will randomly vary. Next, PED.*phiVariance* allows us to change the right-left pointing of our ejection vector, but we can only adjust the right direction of the ejection vector. Left is always stuck at 0 degrees. Finally, PED.*phiReferenceVel* can be used to cause the emitter to spin clockwise about its up vector at N degrees per second.

[5] The theta and phi parameters ought to offer the same features, with theta for up-down and phi for right-left. Both should have a min/max range, and both should be able to spin the vector.

Orientation Explained

OK, we've covered orienting the ejection vector, but what about the particle itself? Well, first remember that the particle is actually a billboard. Initially, I said that these billboards will 'normally' face the camera. The PED orientation parameters give up the ability to choose between various billboard orientations. The following table summarizes particle orientation options.

PED.orientParticles	PED.orientOnVelocity	Resulting Orientation
false	don't care	Screen Oriented – Particle always faces screen (camera).
true	false	Face Ejection – Face along ejection vector.
true	true	Face Motion – Face along trajectory.

Animated Textures

Among the other cool features supported by Torque's Particle Engine, is the ability to animate a particle via multiple textures. The concept is a familiar one which you probably see every time you surf the web. You will often see animated buttons, images, etc. These are often made with animated GIFs. Basically, the image file contains several different images which are displayed in rapid order, over and over.

So, how does this apply to Torque? Well, in Torque you can specify up to 50[6] separate textures. Then, while the particle is being displayed, Torque will cycle through these images.

OK, great! But how do we do this? It's really quite simple. Take a look at the following example:

```
PD.animTexName[0]  = "~/path_to_texture/texture0";
PD.animTexName[1]  = "~/path_to_texture/texture2";
...
PD.animTexName[49] = "~/path_to_texture/texture49";

PD.framesPerSec    = 1; // Play one frame per second
```

In the above example, we've specified 50 distinct textures for use in our sequence. Then, we specified that they must be played one (frame) per second. When the sequence gets to the end, it will begin to repeat. Its really that simple.

[6] If you're willing to edit the engine, you can set this value to anything you want (within reason).

Multiple Particles?

The observant readers will recall that we could specify more than one particle for the PED.*particles* parameter. If you specify multiple particles for an emitter's PED, the emitter will eject the particles in order and then repeat. The reasonable questions that follow are:

1. How do I specify more than one PD, and
2. How many can I specify?

Here are three examples of the syntax for specifying three particles for a PED:

```
particles  = PD_Name0 TAB PD_Name1 TAB PD_Name2; // OR
~~
particles  = PD_Name0 SPC PD_Name1 SPC PD_Name2; // OR
~~
particles  = "PD_Name0   PD_Name1   PD_Name2";
```

Basically, PED.*particles* needs to be a whitespace separated string of PD names. As to how many you can have? Well, I've tried as many as 22 just for fun and this seemed to work, so I'm guessing it will support as many as you will need. However, experimentation will tell. Note, if you do hit a limit, try making the string shorter by shortening PD names. There may be a limit of 255 characters for the PED.*particles* string.

Holy Popping Particles Batman!

An interesting problem I initially had while playing with particles was a disturbing 'popping' effect when the particles' PD.*lifetime* limit was hit. This can have several sources, but if you study the effect it should be apparent that the cause is simply the fact that a very visible object is suddenly popping into non-existance.

So, how do we make this transition more subtle? Simple, just use the particle interpolation parameters to your benefit. Here are some suggestions:

- Be sure your interpolations are smooth. i.e. Don't use values like: 0.1, 0.5, 0.6, 1.0, unless you are looking for a shuddering effect.
- Fade particles by lowering the fourth PD.*colors* parameter (which represents intensity or alpha) over the lifetime of the particle.
- Shrink particles in the latter part of their life.

Sample Emitters

EFM - INCOMPLETE

Advanced Topics

Once you've mastered the basics of particle emitters, you'll likely come up with lots of cool things you can't quite figure out how to do. For example,

- Particle Emitters attached to objects
- Particles that collide with and rebound from objects and terrain
- Moving Particle Emitters
- Particle Emitters triggered by events

This is why there is an OJT section to the guide. We'll revisit Particles in an OJT chapter and explore the above ideas as well as other particle topics. For now, lets move onto the fx Objects.

fxShapeReplicator & fxFoliageReplicator

These two replicators are birds of a feather, both created by Big Dog Melvin May. Their purposes are multi-fold, to include:

1. Allowing multiple objects to be placed automatically and randomly within specified bounds.
2. Allowing this to be done in such a way as to make the scene look more organic (i.e. non-artificial).
3. Reducing the network transmission cost of multiple related objects to that of a single object plus a few additional parameters.

Melv has managed to do this quite successfully, very-much to the appreciation of Torque owners. Furthermore, his fx Objects are, for the most part, easy to understand and use.

Before we get into the usage of these two replicators, I'll give a succint list of all parameters for both the fxShapeReplicator and the fxFoliageReplicator. To save space and due to the common nature of these replicators, I'll combine their parameters into one list, giving indication when a parameter exists in the shape replicator but not the foliage replicator, or vice versa.

Replicator Features

Some of the replicator features that Torque supports are:

- Directed Random Placement – Using a tricky inner- and outer-ellipse affordance, you can direct Torque to replicate a specific number of objects in random locations within a clearly defined area.

- Multiple Toggle-able Placement Restrictions – Because random placement wouldn't be any good if you couldn't specify rules for where to place and not to place, the replicator mission objects both a slew of toggle-able tests for placing objects.

- Dimension and Orientation Controls – In order to make scene more organic, you can provide metrics which will allow objects to be randomly sized and oriented within set bounds.

- Advanced Culling – The Foliage Replicator Provides the ability to tune the culling algorithm. The culling algorithm is responsible for choosing when to render objects and directly affects frame-rate. The ability to fine tune this is a real plus.

- Animation and Lighting – Foliage can be both animated and lit (or self-lit). You have direct control over how this is done.

Placing Replicators

Replicators are placed much like any other item in the world. You just drag them and drop them where you wish the to be. The location of the replicator can be the center of a placement target. The size and shape of this target are controlled by the inner and outer radius parameters. These parameters can be used to create two ellipsoidal areas. If we ignore restrictions for a moment, placement rules simply become:

Position	Can Place Here?
Inside area defined by *InnerRadiusX* and *InnerRadiusY*?	Yes
Inside area defined by *OuterRadiusX* and *OuterRadiusY*, **and** Outside area defined by *InnerRadiusX* and *InnerRadiusY*?	Yes
Outside area defined by *OuterRadiusX* and *OuterRadiusY*?	No

Replicator Visual Feedback

Alright, that seems pretty simple, but how do we see where these ellipses are? Fortunately, Melv has supplied a nice visual feedback mechanim (which can be turned off).

Examining the image above, we can see two ellipses which were created with these settings:

InnerRadiusX == 5
InnerRadiusY == 15

OuterRadiusX == 25
OuterRadiusY == 20

If you look closely, you will see that objects are randomly placed in the area outside inner ellipse and inside outer ellipse.

Seeds

A very important aspect of replicators is that they will produce the same result each time they are used as long as they are given the same *Seed*. The *Seed* is used as an input to a random number generator. This generator is used to produce and place all objects associated with the replicator.

Replicant Count

You may select how many objects you wish to replicate using either the *ShapeCount* or the *FoliageCount* parameter, depending upon which replicator you are using. It is important to understand that this is a theoretical maximum, not the guaranteed number of objects you will get. Besides the ellipses and the position, what else controls placement?

Placement Restrictions (Restraints)

Melv is no stranger to this kind of work and his experience shows through. He has supplied a nice set of 'knobs' with which we can tune placement rules. These are called Restrictions or Restraints in the foliage and shape replicators respectively. Their names are pretty self-explanatory, but just in case I'll explicity spell out their use here:

Restriction / Restraint	Result
AllowOnTerrain	If this is set to true, objects can be placed on terrain if present.
AllowOnInteriors	If this is set to true, objects can be placed on interiors (buildings, etc) if present.
AllowOnStatics	If this is set to true, objects can be placed on other shapes if present. This means that if you are using the fxShapeReplicator, it is possible to have objects get stacked on top of each other by a replicator. See image below.
AllowedTerrainSlope	When objects are placed on terrain, they will not be placed on areas with a slope equal to or greater than this value.

AllowOnWater	*AllowWaterSurface*	*AllowOnTerrain*	Result
false	-	-	Objects cannot be placed in areas with water
true	true	-	Objects can be placed on surface of water.
true	false	true	Objects can be placed on terrain below water

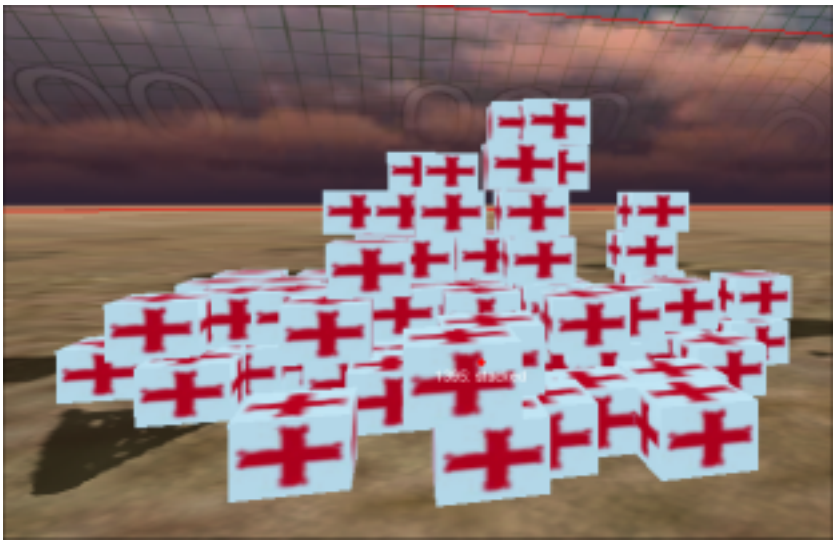


Fig X. Stacked Shapes using fxShapeReplicator (*AllowOnStatics* == true)

In addition to the above Restraints, fxShapeReplicators offer three additional parameters. *AlignToTerrain* causes shapes that are placed on terrain to align to the terrain's up vector. Furthermore, you can adjust how this alignment occurs by adjusting the parameter *TerrainAlignment* which is a 3 value vector. Lastly, you can enable or disable shape collision boxes by setting *Iteractions* to true or false respectively.

Retries

Well, with all these rules determining whether an object can be placed, you must wonder what the replicator does if it finds it can't place an object. Well, just like you or I, it tries again. You can control the number of attempts the replicator will make per object with the *FoliageRetries* or the *ShapeRetries* parameter. "Why not just try until an object can be successfully placed", you ask? Consider the case where there is no legal place to put an object left. In this case, without a retry limit the replicator would attempt to place objects forever…

Foliage Dimensions

Alright, we've finished talking about the common attributes between the fxFoliageReplicator and the fxShapeReplicator. Now lets jump into some of the additional features offered by Foliage. Because we're going to be using the same image over and over to simulate some kind of foliage feature, we'd like an inexpensive way to make these images 'seem' different. The Dimension parameters give us this. For example, lets say we choose the following settings:

FixSizeToMax == false	*MinWidth* == 0.5	*MaxWidth* == 1.5

FixAspectRaio == false *RandomFlip* == true	*MinHeight* == 0.5	*MaxHeight* = 2.0

What we would get would is billboards that are randomly between 0.5 and 1.5 times their default width and 0.5 and 2.0 times their default height. Additionally, the image may be randomly flipped around it vertical axis (i.e. flipped horizontally). This flipping will be useful if we have a non-symmetric image. So, what about that aspect ratio business? Well, if you are familiar with texture mapping you will understand that without maintaining the proper aspect ratio images may look stretched. The *FixAspectRatio* forces the randomly selected height/width to be a fixed multiple of the original. Here are some example images to show what I'm talking about:

128 x 128 PNG

Same PNG 2X Height
FixAspectRatio == false

Same PNG 2X Height
FixAspectRatio == true

Lastly, lets discuss *OffsetZ*. This is helpful to fix little issues you run into where the texture may be slightly embedded or slightly above a surface. If this happens, just increase or decrease *OffsetZ* slightly till the problem is fixed.

Shape Dimensions and Rotation

fxShapeReplicators allow you to adjust the dimension and rotation of shapes with the parameters in the Object Transforms group. You can allow random scaling by setting *ShapeScaleMin* and *ShapeScaleMax* accordingly. Additionally, you can allow for random rotation by setting *ShapeRotationMin* and *ShapeRotationMax* to non-zero values. Values are chosen between Min and Max on a per-axis basis. Finally, *OffsetZ* is offered under the group for fxShapeReplicators and has the same purpose and noted above.

Foliage Culling

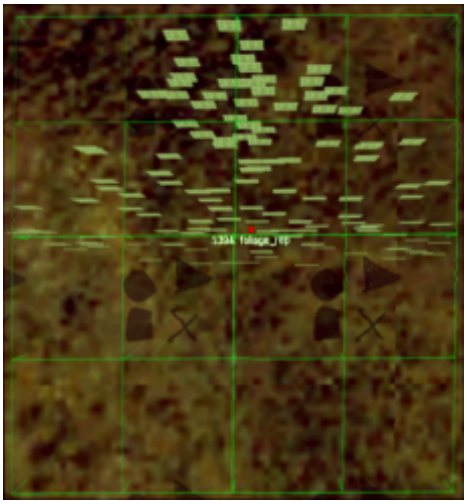
Of all the attributes in the fxFoliageReplicator, the culling parameters were the least intuitive to me. Before we jump into them, perhaps a quick description of view culling is in order.

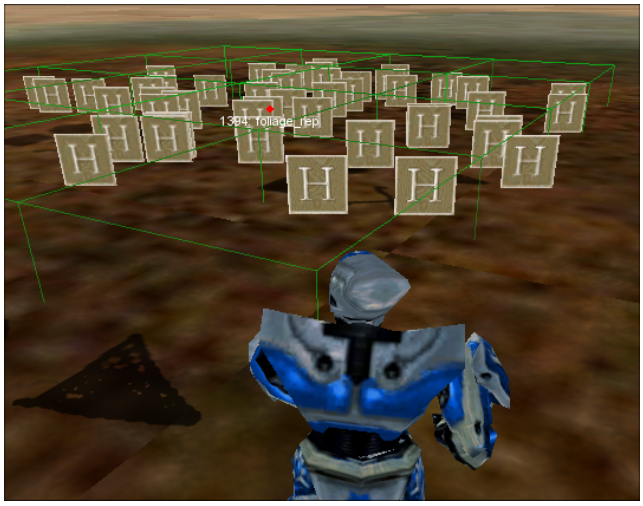
View Culling

If you think abou it for a moment, it will be apparent that it would be highly inefficient to render all objects in a mission, when only a small fraction of them are in a position to be visible. In reality, the objects in front of the camera are the only objects that we really need to render. This set of objects is called the Potentially Visible Set (PVS). There are many ways to build a PVS. In the case of fxFoliageReplicators, when the *useCulling* parameter is false, each billboard is individually tested for visibility. In the case of a small set of billboards this is probably the most efficient way to cull. However, once you have a large number of objects this method quickly begins to consume too much CPU time.

Quad-Culling

At this point, you should consider turning on culling by setting *useCulling* to true. Now, culling is tested against a set of quads instead of individual billboards. A quad is a rectangular area (usually a square) with a fixed dimension. In the case of quad-culling, a specified area is subdivided into multiple quads. Each object that is within a the area defined by a quad is algorithmically associated with that quad. Objects that cross borders between quads are assigned to each quad they touch. Finally, if a quad is deemed to be visible, all objects associated with that quad are marked as visible and susbsequently rendered. The images below are taken from an in game shot to demonstrate what the visible feedback for quad culling looks like. As well, they demonstrate the discussion thus far.





Configuring (Quad) Culling

I'm sure that this is all just facinating, but it still leaves us with the dilema of how to choose whether to cull and then if we choose to cull, how to set up our culling. Unfortunately, the number of factors involved turn this more into an art than a science, and the final test is always going to be frame rate. However, I'll supply some rules of thumb to help you out in your choice.

To Cull or Not To Cull:

- Do not use culling for small sets (1-100) of billboards.
- Generally it is better to use culling if total quads number at least 2-3 times fewer than billboards (accounts for overhead associated with algorithm).
- For a large number of objects (100's to 1000's), spread over a large area (1/4 of map or more) it is best to use culling.
- Culling will not help much if your objects are not evenly distributed between the quads.

Selectting a CulResolution:

- Select your *CullResolution* such the number of quads comes out to at least 2-3 times fewer quads than objects.
- Select your *CullResolution* such that it can evenly divide *OuterRadiusX* and *OuterRadiusY*. You may need to adjust these slightly to assist this process. Powers of 2 are nicest if possible.

Testing Efficiency of Culling

As noted above, the best way to test the efficiency of your culling is to check your average FPS. An easy (if possibly slightly inaccurate) way of doing this is by,

1. Get out of Mission Editor Mode.
2. Start the console (~).
3. Type: metrics(fps);
4. Exit the console (~)
5. Walk/Fly around your scene and observe your FPS. Look for hot-spots where it dips.

The '*metrics(fps);*' command will create a GUI in the upper left corner of the screen, showing FPS and mspf (milliseconds per frame). This GUI will be shut off when you start the Mission Editor and does not render properly while it is running.

Additionally, after hitting apply (when setting your culling parameters), you can get additional data from the console (~). Each time you hit something like this is printed in the consolse:

```
fxFoliageReplicator - Lev: 3 PotNodes: 85 Used: 58 Objs: 656 Time: 0.0160s
fxFoliageReplicator - Approx 0.06Mb allocated.
```

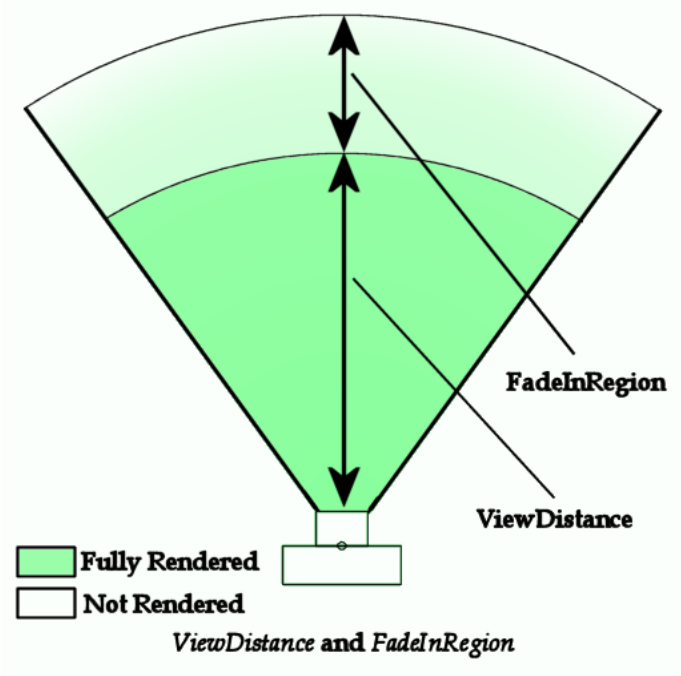
From this we can see that the culling Level is 3 which means it is a 2^3 x 2^3 (8 x 8) set of quads. The quads are approximately 58/85, or 68% utilized (i.e. billboards are in 68% of the testable nodes). There are a total of 656 objects (500 billboards and 156 phantom objects due to retries[7]). It takes about 160ms to build and render the fxObject. And, finally, the entire fxObject takes up about 0.06 MB. Pretty impressive eh?

[7] This is a guess actually. Melv, if you are reading this and find I've described any of your fxObjects wrongly, please let me know and I'll amend this.

Other Culling Features

In addition to quad culling, there are some other features in the culling parameters section, specifically the view, fade, and alpha parameters. These parameters are not affected by the *useCull* parameter and are always ON.

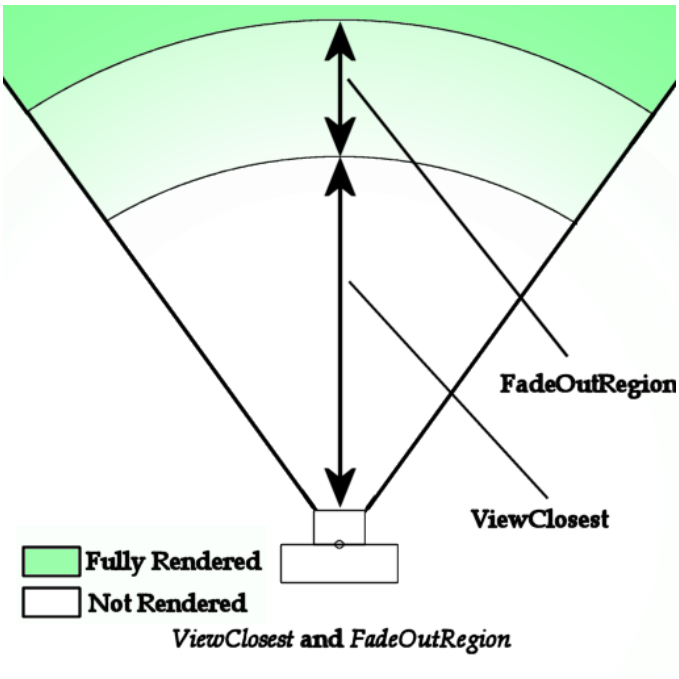
ViewDistance and *FadeInRegion* work together to determine when an object begins to fade into view and when it is fully faded in. These two parameters form concentric spheres around the camera, where *ViewDistance* defines the radius of the inner sphere and *FadeInRegion* + *ViewDistance* defines the radius of the outer sphere. When an object is at the perimeter of the outer sphere it will begin to become visible, fading completely in at the perimeter of the inner sphere. If you wish your objects to stop rendering at an alpha greater than 0.0, you can cause this to happen by setting *AlphaCutoff* to the desired alpha, betweem 0.0 and 1.0.



ViewDistance and *FadeInRegion*

Billboard's Distance to Camera	Render?
Distance > *ViewDistance* + *FadeInRegion*	NO
ViewDistance < Distance < *ViewDistance* + *FadeInRegion*	YES (if Alpha > *AlphaCutoff*)
ViewDistance < Distance	YES

ViewClosest and *FadeOutRegion* also work together, but their effect is the opposite of *ViewDistance* and *FadeInRegion*. Conversely, these two parameters are used to determine when an object begins to fade out of view and then become fully transparent or not rendered. Again, these two parameters form concentric spheres around the camera, where *ViewClosest* defines the radius of the inner sphere and *FadeOutRegion* + *ViewClosest* defines the radius of the outer sphere. When an object is at the perimeter of the outer sphere it will begin to fade, fading completely out at the perimeter of the inner sphere.



ViewClosest and *FadeOutRegion*

Billboard's Distance to Camera	Render?
Distance > *ViewClosest* + *FadeOutRegion*	YES
ViewClosest < Distance < *ViewClosest* + *FadeOutRegion*	YES (if Alpha > *AlphaCutoff*)
ViewClosest < Distance	NO

You may wonder why you would want to do this. Consider the case where you are in a vehicle. Fading out will keep objects from suddenly being inside the vehicle.

Lastly, I'll mention *GroundAlpha*. This parameter can be used to force the bottom of billboards to have a lower alpha value. This can be used to moderate the harsh intersection between bilboards and the ground, giving the transition a cleaner look. Just set it to a value lower than 1.0 to see its effect. Adjust it till you are pleased with the end result.

Foliage Animation

Foliage Animation is a feature which allows us to make the a more interesting and convincing scene. Consider the case where your foliage is long grass and fronds. Wouldn't it be more realistic if the grass and fronds blew in the wind? Yes, of course it would be. But how do we achieve this look? With foliage animation of course!

Setting *SwayOn* to true will enable the animation. you may cause your billboards to sway side-to-side or and front-to-back, using the *SwayMagSide* and *SwayMagFront* parameters respectively. Furthermore, you can add a little spice to the swaying by allowing the sway times to vary between *MinSwayTime* and *MaxSwayTime*. Lastly, you may choose to enable *SwaySync*, where all objects will sway together in the same way, or you may disable it and all objects will sway on their own pattern.

One word of caution. If billboard sway so much that they touch each other, you will get rendering artifacts.

Foliage Lighting

Foliage Lighting is the last parameter group we will discuss. It is another group that is used to make the scene look more interesting. With these parameters, you may enable self-lighting (*LightOn*). Furthermore, if you set *LightSync* to false and give different values for *MinLuminance* and *MaxLuminance* each billboard will be self-lit with its own randomly selected level of light. Please note that this lighting can be animated. If all of the above lighting parameters are set as noted and then you set *lightTime* to a non-zero value, each billboard's lighting will vary over time. *lightTime* is the time for a fade in one direction. So, to fade from *MaxLuminance* to *MinLuminance* back to *MaxLuminance* will require (*lightTime* * 2) seconds.

fxSunLight

As previously mentioned, the Sun object controls scene lighting and fxSunLight provides the ability to have a visible sun(s) in the sky. Upon first inspection, this mission object may seem a bit daunting, with it myriad of parameters (lerps, animations, etc), but never fear, it is really quite easy to use. You've really got to hand it to Melv though. He hardly makes a resource without a 'few' options.

fxSunlight Features

Some of the fxSunlight features that Torque supports are:

- Two Flare Types – fxSunlight supports both a local flare (respresenting the lens flare of a camera), and a remote flare (representing the sun object itself). Both flares are configurable.

- Position Parameters which Match Sun Objects – To make life easy, the fxSunlight parameters which control it's position are similarly named to those found in the Sun mission object. Namely Azimuth and Altitude.

- Animations – Just about every characteristic of the fxSunlight Object is animate-able. Furthermore, the animation system is a very flexible key based animation system.

Adding a New fxSunlight

1. Start the Creator,
2. Find and click 'Mission Objects → environment → fxSunlight'
3. Enter a name for this Sun in the pop-up box. Ex: 'Smiley'
4. Click OK.

At this point, if you look around, you should see the default fxSunlight. Now,

5. Switch to the Inspector.
6. Locate your new sun (Smiley).
7. Select it.

Changing the Sun Images

fxSunlight has two texture parameters,

- *Media→LocalFlareBitmap*
  - This texture represents a lens flare effect.
  - If you do not wish to have this effect, just clear this parameter.

  - This texture will render if it is in line-of-sight. If it is blocked by terrain or an object, it stops rendering.
  - It is best to use a texture with an alpha layer.
- *Media→RemoteFlareBitmap*
  - This texture represents the sun itself.
  - It too can be disabled, just by clearing this parameter.
  - Unlike the local flare, this texture renders all the time although it can be occluded by the terrain and objects.
  - Again, it is best to use a texture with an alpha layer.

Note, you should make both textures the same way. That is, if one has an alpha layer, the second one should too.

Positioning the Sun (Render Position)

Unlike most mission objects, the standard position, rotation, and scale are meaningless and do not control where the fxSunlight object is rendered. However, there is a marker at *Transform→position*. I would just select value for this such that the marker does not get in your way while editing.

Render Position, when it is not being animated, is based on the same two concepts as those used for the Sun Object, azimuth and elevation. If you do not understand these concept, I suggest you quickly re-read the Sun object description above.

- *SunOrbit→SunAzimuth*
  - This controls the horizontal angle of the fxSunlight effect's bearing about the Z-axis.
  - Legal Values: [0, 360)
  - Make this the same as Sun→*Misc→azimuth*.

- *SunOrbit→SunElevation*
  - In simple terms, this controls the 'elevation', but in reality this is a polar angle. Again, if you don't understand this, see the Sun object description.
  - Legal Values: [-90, 90]
  - Make this the same as Sun→*Misc→elevation*.

Changing Lens Flare Effects

You can modify various effects such as:

- *LensFlare→FlareTP*
  - If this is not checked, the lens flare will not render in 3rd POV.

- *LensFlare→Colour* (R G B I)
  - If you find a white lens flare boring, you can give it a different fixed color with this parameter.

  - Each individual value can between 0.0 and 1.0.
  - Intensity has no effect.

- *LensFlare→Brightness*
  - You can set a fixed brightness with this parameter.
  - Legal values: [0.0, 1.0]

- *LensFlare→FlareSize*
  - This parameter can be used to scale the flare size to your preference.
  - This modifies the size of the sun too.
  - Legal values: (0.0, inf)

- *LensFlare→FadeTime*
  - This parameter determines how long it takes the lens flare to fade away when it is occluded. Remember, occlusion turns it off.
  - Legal values: [0.0, inf)

- *LensFlare→BlendMode* – Understand that the flare is rendered, meaning it needs to be blended with the prior contents of the framebuffer. To accommodate various effects, fxSunlight support three blending modes [0, 2]:
  - 0 – glBlendFunc(GL_SRC_ALPHA, GL_ONE)
    - Flare <R G B A> replaces framebuffer <R G B A>.
  - 1 – glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
    - <1 1 1 1> - Flare <R G B A> replaces framebuffer <R G B A>.
  - 2 – glBlendFunc(GL_ONE, GL_ONE)
    - Flare <R G B A> is Added to framebuffer <R G B A>.

If you stopped right now, you would know 90% of what you need to know about the fxSunlight object. However, if you want to do some really cool things, like animate the color, brightnes, and size. Or if you want it to rotate and to move around over time, then by all means, continue reading.

Animating The Sun and Lens Flare

Now that we have a fxSunlight object set up, we can make it more interesting by animating some of the Sun and Lens Flare effects. However, before we take a brief tour of the fxSunlight animations, lets discuss some common animation parameters.

Anim*

If this parameter is checked, the animation is enabled.

Lerp*

Checking this parameter enables LERPing. When LERPing is enabled, all values are linearly interpolated between key-frames. In other words, transitions are smooth. If LERPing is disabled, transitions are sharp and may pop/flash between values.

Min* and Max*

You can specify min and max parameters for all animations. These values vary based on the animation type. See below for specifics.

***Keys**

*Keys may be a little confusing at first. The *Keys parameters take a variable length text string containing letters between 'A' and 'Z'. A value of 'A' means the animation is at the Min* setting for this effect. 'Z' means it is at the Max*. Here are some examples:

*Keys String	Meaning
A	Stay at the minimum value for entire animation.
AZA	Start at minimum value, transition to maximum value, then transition back to minimum value.
ACBEDGFIHKJMLONQPSRUTWVYXZA	Start at minimum value and transition to the maximum value, but oscillate back and forth on the way. Upon reaching the maximum value, quickly transition back to minimum.

***Time**

As you might imagine, there needs to be a way to determine the period of the animation. The *Time parameters do this. *Time is the number of seconds it takes for the animation to cycle once.

Now, onwards to the animations…

Colour Animation

- *Animation Options →AnimColour*
- *Animation Options →LerpColour*
- *Animation Options →SingleColourKey*
  - Instead of using individual keys for Red, Blue, and Green, tie all colors to the *RedKeys* parameter.
- *Animation Extents →MinColour*
  - Each of the first three elements of this parameters may take values:
    - [0.0, *MaxColour*]
  - The fourth element should be 1.0.
- *Animation Extents →MaxColour*
  - Each of the first three elements of this parameters may take values:
    - [*MinColour*, 1.0]
  - The fourth element should be 1.0.
- *Animation Keys →ColourKeys*
- *Animation Options →ColourTime*

Brightness Animation

- *Animation Options →AnimBrightness*
- *Animation Options →LerpBrightness*
- *Animation Options →LinkFlareSize*
    - If this is checked and brightness is animating, the size of the Lens Flare bitmap is proportional to the current brightness.
- *Animation Extents →MinBrightness*
    - This parameter may take a value [0.0, *MaxBrightness*]
- *Animation Extents →MaxBrightness*
    - This parameter may take a value [*MinBrightness*, 1.0]
- *Animation Keys →BrightnessKeys*
- *Animation Options →BrightnessTime*

Rotation Animation

- *Animation Options →AnimRotation*
- *Animation Options →LerpRotation*
- *Animation Extents →MinRotation*
    - This parameter may take the value [0.0, *MaxRotation*].
- *Animation Extents →MaxRotation*
    - This parameter may take the value [*MinRotation*, 360.0)
- *Animation Keys →RotationKeys*
- *Animation Options →RotationTime*

Size Animation

- *Animation Options →AnimSize*

- *Animation Options →LerpSize*

- *Animation Extents →MinSize*
    - This parameter may take the value [0.0, *MaxSize*]

- *Animation Extents →MaxSize*
    - This parameter may take the value [*MinSize*, inf)
- *Animation Keys →SizeKeys*

- *Animation Options →SizeTime*

Azimuth Animation

- *Animation Options →AnimAzimuth*
- *Animation Options →LerpAzimuth*
- *Animation Extents →MinAzimuth*
    - This parameter may take the value [0.0, *MaxAzimuth*]
- *Animation Extents →MaxAzimuth*
    - This parameter may take the value [*MinAzimuth*, 360).
- *Animation Keys →AzimuthKeys*
- *Animation Options →AzimuthTime*

Elevation Animation

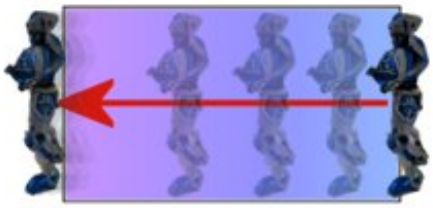
- *Animation Options→AnimElevation*
- *Animation Options→LerpElevation*
- *Animation Extents→MinElevation*
    - This parameter may take the value [-90.0, *MaxElevation*].
- *Animation Extents →MaxElevation*
    - This parameter may take the value [*MinElevation*, 90.0].
- *Animation Keys→ElevationKeys*
- *Animation Options→ElevationTime*

Physical Zone

Physical Zones are one of those simple, “Gee Whiz! Now ain’t that cool,” kinds of constructs. In fact, of all the standard torque mission objects these are probably my favorite. Physical Zones, or P zones for short, allow you to define areas in your game with modified gravity and/or velocity modifiers and/or an applied force. Eh? Well, hold that thought while we cover the very few parameters P zones have, then we’ll leap right in.

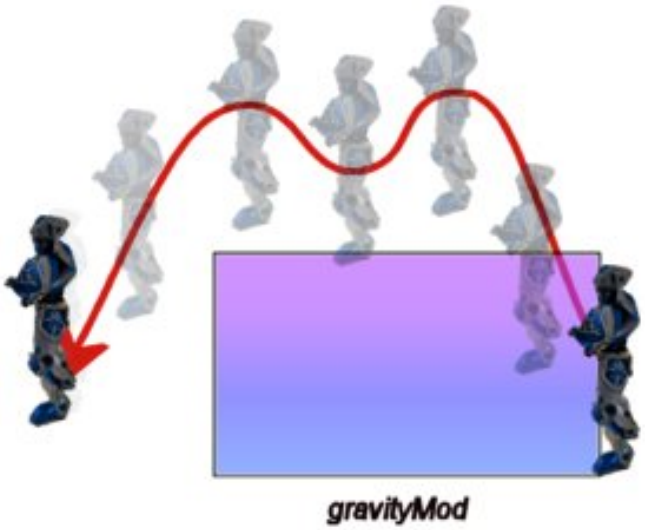
velocityMod

The *velocityMod* attribute does pretty much what it sounds like it will do. Let’s say we have a P zone with a *velocityMod* of 2. If the player enters the P zone with a velocity of 10.0 m/s s/he will leave the zone with a velocity of 20.0 m/s. Actually, the velocity mod is instantaneous, occurring directly after entering the P zone. It should be noted that there are some issues with extraordinarily high *velocityMod* values. If the multiplier is too high, the engine can freeze for long periods or even crash. So, my suggestion is to keep the values low while you experiment. The upper bounds of [-40.0, 40.0] are really too high for most practical uses.



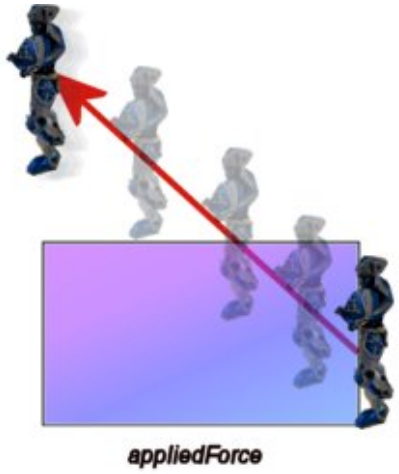
gravityMod

The *gravityMod* attribute specifies a local (area inside P zone) gravity multiplier. In other words, if *gravityMod* is –2 and the game gravity is set to 1.0, then when the player enters the P zone s/he will float upward. If the player has enough forward velocity upon entering the P zone, s/he will end up 'skipping' across the P zone till s/he fall off the end or encounters an obstacle. Be careful with NULL or negative gravity zones. If the player gets stuck with his feet off the ground, he will be unable to move. Again, high values can cause problems for the engine. Caution is the word.



appliedForce

Finally, the *appliedForce* vector. This attribute allows you to create an area where an invisible force will be applied to the character. This force can point in an arbitrary direction with a variable strength.



Here is a table of values and their effects on the character while on a flat surface:

0 .. 99	100-399	400-1999	5000	40000
Practically no movement.	Sorta slides along.	Forced walk	Forced run	Can you say cannon?

fxLight

The purpose of this object is to provide dynamic lights.

This chapter is incomplete, but for now please realize that most of the features for this object behave and are configured like the fxSunlight object.

fxLight Features

Some of the fxLight features that Torque supports are:

- EFM - INCOMPLETE.

Path

This chapter is incomplete.

PathMarker

This chapter is incomplete.

Trigger

This chapter is incomplete.

Camera

This chapter is incomplete.

SimGroups

This chapter is incomplete.

