David Sushil davidjsushil@gmail.com

Particle Effects in Torque

Introduction

Before we begin, go put some coffee on. You'll need it later.

In modern 3D games, designers are often faced with the challenge of balancing strain on the system with pure gloss. As our cowboy hero rides across the desert, we expect little clouds of dust to appear beneath the metal-shod hooves of his trusty steed. We expect the heat of the desert to warp and wobble the sight of distant mountains. We expect flies to buzz about the ears of buffalo, and shell casings to fall from discharged rifles.

Clouds of dust and swarms of flies are difficult to model in 3D, and as 3D models, they would hog system resources. As a result, they must be faked. For this, we implement particle emitters. These special effects systems are comprised of sprites (sometimes called billboards), which are bitmap images with some level of transparency. They may be animated. In any case, they almost always face the screen.

Creating particle emitter systems in Torque is really a three step process. It requires creating the particles themselves, the behavior that drives the particles, and spot for them in 3D space. These are manifested in Torque as Particle Data, Particle Emitter Data, and Particle Emitter Node Data. If these terms scare you, don't sweat it – once we have a bit of working code on screen, it will all make sense.

Particle Data (PD)

We will begin by looking at particles. These could be sparks, puffs of smoke, drops of water, or specks of dust. They have certain characteristics, like transparency, a life expectancy, and how they are affected by wind and gravity.

To create a Torque particle, we begin by making the image in a program like Photoshop or GIMP. Particle image sizes should be powers of 2, not exceeding 512 x 512 pixels. Keep in mind that Torque ignores all color data in this image, and uses it instead as a transparency map. In this way, black pixels will be transparent; white pixels, opaque; gray pixels, translucent. Save your image as either a JPEG or PNG file.

Creating a particle datablock in Torque is fairly simple, once you've done it once or twice. Let's look at some sample code:

```
datablock ParticleData(steam_puff){
      textureName = "~/data/shapes/steam/steam_puff.png";
      lifetimeMS = 300;
      lifetimeVaranceMS = 100;
      colors[0] = "0.5 0.5 0.5 0.75";
      colors[1] = "0.4 0.4 0.45 0.5";
      colors[2] = "0.3 0.3 0.4 0.125";
      colors[3] = "0.2 0.2 0.35 0";
      sizes[0] = 1.0;
      sizes[1] = 1.1;
      sizes[2] = 1.5;
      sizes[3] = 2;
      times[0] = 0.0;
      times[1] = 0.3;
      times[2] = 0.6;
      times[3] = 1;
      spinSpeed = 0;
      spinRandomMin = -100;
      spinRandomMax = 100;
      gravityCoefficient = -10;
      constantAcceleration = 5;
      dragCoefficient = 2;
};
```

Here, we create a ParticleData object template and call it steam_puff. You will notice that the first property within our datablock is a texture name, which points to our image file. When using Torque, it's not necessary to include a file extension such as .PNG or . JPG. Torque will instinctively try to open either one.

Next, we have lifetime information, which indicates how many milliseconds a particle may remain in the world. In this case, our steam puff particle will last three tenths of a second, give or take 100 ms, as indicated by the lifetime variance property. Creating variability is an excellent way of giving your particles some character. By the way, how's that coffee?

Next, there are lines which control the color and size of the particle throughout its lifetime. Notice the lines that begin with times[0] – times[3]? These are keyframes, and the values they contain represent percentage values, with 0 being the instant the particle enters the world, and 1 being the instant that it leaves. Torque will look at the information contained in the colors and sizes arrays, and interpolate between them based on the times values.

The colors array accepts four values for each member, which represent red, green, blue, and alpha values. Remember, our particle image is a transparency map, and otherwise contains no color, so we have to colorize it ourselves. For color values, 0 always represents no color, while 1 represents full color. Likewise, 0 is invisible, while 1 is fully opaque. In this example, or particle begins life as a semi-transparent gray (red, green and blue are at 50% each, and our alpha channel is at 75% opaque). Over time, it becomes darker and darker, and more and more transparent. If you notice the blue channel, it grows darker slightly less than the red and green channels, so our steam puff also appears bluer as time goes on.

The sizes array works exactly like the colors array, with the size of the particle changing based on the values stored. A value greater than 1 indicates an increase in size, which a value less than 1 indicates a decrease in size. For our steam puff, it doubles in size from its initial appearance to its untimely death. Please note that the members of the sizes array can take values ranging from 0 to infinity.

Now, do you have to include four keyframes? No. In fact, you don't have to include any, but if you don't, Torque defaults to solid white sprites exactly the size you created them, which don't change their appearance from the time they're born to the time they die.

There are three variables that control the rotation of a particle. They are spinSpeed, spinRandomMin, and spinRandomMax. The spinSpeed property can accept values from -10,000 to 10,000. In this case, we indicate no spin at all, which is 0. On the other hand, we specify a spinRandomMin of -100, and a spinRandomMax of 100, which allows Torque to randomly adjust the spin of each particle. Again, this adds variability, and variability adds character.

The final three properties control how a particle's motion changes over time. The first property, gravityCoefficient, indicates how gravity affects each particle. Particles with a positive gravity coefficient will fall, while those with a negative coefficient will rise. Since we're working with steam, we want it to rise, hence the negative value. The constantAcceleration property indicates how fast the particle moves. It can accept either positive or negative values, indicating forward or backward motion, respectively. Finally, the dragCoefficient property indicates how much the particle is slowed per second. We don't have to worry about this much, since our steam has a shelf life less than one second. However, it's useful to demonstrate how particles can be slowed with time.

Are these all the properties for a particle? No. There are more, but these are enough to get you started.

Now, a particle on its own does nothing. In order to generate a stream of particles, we need a particle emitter, which, coincidentally enough, is what we cover in the next section.

Particle Emitter Data (PED)

Particle emitters handle how our particles come into the world, how fast they travel, and how they move. Particle emitters also have properties which control how long they last, and which way our sprites face.

Taking a look at an effective emitter for steam, we find:

```
datablock ParticleEmitterData(steam_column){
      particles = steam_puff;
      ejectionPeriodMS = 25;
      periodVarianceMS = 2;
      ejectionVelocity = 5;
      velocityVariance = 2;
      thetaMax = 10;
      thetaMin = 0;
      phiReferenceVal = 0;
      phiVariance = 10;
      lifeTimeMS = 0;
};
```

This particle emitter is called steam_column. You'll notice that the first property, labeled particles, accepts our steam_puff ParticleData template. This means steam_column will generate multiple steam_puffs, based on the remaining properties in its declaration.

First, we have properties which control how our steam puffs are brought into the world. In this case, ejectionPeriodMS indicates how often a particle is emitted, and periodVarianceMS gives us some variability. So our column of steam will generate one puff of steam approximately every 25 milliseconds. Rather frequent, but hey, it's steam.

Next, ejectionVelocity and velocityVariance control how fast each particle travels away from the emitter. Don't get confused between ejectionVelocity in the emitter and constantAcceleration in the particle itself. These two motion-based properties could be forcing the particle along two different paths – say, up and right.

In fact, that's where thetaMax and thetaMin come into play. These properties essentially control the emitter's aim, at least around the x-axis. They can accept a range from 0 to 180, but of course, the minimum or maximum depends on its counterpart's value. So, if thetaMin is 10, thetaMax must be at least 10. In any case, a value of 0 sends particles shooting straight up, while a value of 180 sends them straight down.

Likewise, phiReferenceVal and phiVariance control rotation around the z-axis, and can each take a range of values from 0 to 360. Yes, it's rather inconsistent from the theta controls, but it works nonetheless. In this case, our phi reference value of 0 keeps the emitter from rotating around the z-axis, but our phi variance value allows for some wiggle room. Again, this kind of variety adds spice.

Finally, we have a property, lifeTimeMS, which determines how long the emitter itself exists in the world. Don't be fooled, a value of 0 means it never leaves. Any other value for this property must be a positive number. So if you wanted your emitter to work for 5 seconds and then quit, set this value to 5000.

Again, these are not all of the properties for an emitter, but they are enough for now.

Particle Emitter Node Data (PEND)

Finally, we have particle emitter nodes, which contain transformation info, a wink and a nod to our emitter, and an odd little property which affects how the system behaves overall. Here's an example:

```
datablock ParticleEmitterNodeData(steam_source){
     timeMultiple = 1;
};
```

Yeah, it's not much, but this one value affects how often particles are ejected, and their position when ejected. How this happens is not clear to me, so let's assign it the identity value (1), and move on.

Of course, the transformation data and the emitter property aren't assigned within this template. Instead, they are assigned when an instance of the node is created.

Getting Our Steam On

To make the steam particle emitter work, we have to jump through a scant few hoops. Create a file called steam.cs, and place it in demo/data/shapes/steam/. This file should contain all of the datablocks we outlined in the above sections, plus the following code:

```
function addSteam(){
      %steamObj = new ParticleEmitterNode(){
            position = "238 -205.75 191";
            rotation = "1 0 0 0";
            scale = "1 1 1 1";
            dataBlock = steam_source;
            emitter = steam_column;
            velocity = 1;
      };
}
```

Notice, this ParticleEmitterNode object an instance of steam_source, our node template, and its emitter is steam_column. The position, rotation, and scale properties should be more-or-less routine now, and you need only change the position data to reflect your mission. The velocity property indicates an initial ejection velocity for the emitter, exactly why, I couldn't tell you.

This file's ready to go, but if we add it via the console to test it, you'll notice that our pretty steam puff images from page 1 of this document have been hijacked by giant white squares. Something's clearly amiss.

As it turns out, all of our particle effect information must be preloaded into Torque. We can't add it during a mission, or it misbehaves. To preload your particle emitter, locate demo/server/scripts/game.cs, and open it for editing. In the onServerCreated() function,

locate the section set aside for basic data blocks, and right around the reference to chimneyfire.cs (another particle emitter, by the way), add the following line:

```
exec("demo/data/shapes/steam/steam.cs");
```

Now, when you load up Torque, open your console window, and type the following command:

```
addSteam();
```

That was almost as easy as pie, if pie was six pages long, right?

Where To Go From Here

There are only a few additional topics to cover. The first is animating particles. The second is generating multiple particles from the same particle emitter node. The third is moving an emitter once it has been created. All three tasks are possible in Torque. Briefly, let's address them.

Animating Particles

To animate a particle, add the following code to your particle datablock (steam_puff, from page one):

```
animateTexture = true;
framesPerSec = 32;
animTexName[0] = "demo/data/shapes/steam/steam_puff-0.png";
animTexName[1] = "demo/data/shapes/steam/steam_puff-1.png";
animTexName[2] = "demo/data/shapes/steam/steam_puff-2.png";
// Etc., up to animTexName[50]
```

You must set animateTexture to true for this to work. As for the framesPerSec property, 32 is the threshold at which humans see fluid motion, but feel free to choose any value between 1 (the default), and 200.

Multiple Particles

If you want your broken pipe to emit jets of water as well as steam, there's not necessarily a need to create separate emitters. All we need to do is define a new kind of particle, say water_jet, and link it to steam_column, our particle emitter. I'll skip the water_jet particle data template (this tutorial is long enough already), and instead I'll show you how to list multiple particles in the emitter. Check out the code:

```
particles = steam_puff SPC water_jet;
```

This line lists two particle templates, steam_puff and water_jet, which are separated by the keyword SPC, which designates a space.

Moving an Emitter

Normally, particle emitters cannot be moved via code. However, a hack exists which will allow you to move emitters to your hearts content. It goes like this:

```
// Create our particle emitter node, and store its ID in %steamObj
%steamObj = new ParticleEmitterNode(){
      position = "238 -205.75 191";
      rotation = "1 0 0 0";
      scale = "1 1 1 1";
      dataBlock = steam_source;
      emitter = steam_column;
      velocity = 1;
};

// At this point, %steamObj is stuck at 238, -205.75, 191
// However, we could...

%steamObj.position = "100 100 191";
// ...set a new position, which does absolutely nothing, unless...

%steamObj.setScale(%steamObj.getScale());
// ...we use the setScale() function to reflexively assign the scale to
// itself, which has the added effect of changing the position.
// Weird, huh?  But such is Torque.
```

Conclusion

Although this tutorial doesn't cover every nook and cranny of Torque's particle effects capabilities, it's a good overview. The best thing to do now is practice and experiment.