Beginners guide to T3D part two

Lukas Peter Joergensen

August 28, 2013

Revision 1

Introduction

This is part 2 of a beginners introduction to T3D tutorial series I am writing.
In this tutorial we will be extending the coin collection game with a visual effect when you pick up coins, and a quick rewrite of the original code to support muliplayer interaction and an description of what to take in mind when writing games for multiplayer games.

The Author

Who am i?

My name is Lukas Joergensen, I am 19 years old and live in Denmark, currently studying Computer Science at Aarhus University.

What is your experience with T3D?

Currently, my most actual project is the IPS Lite and Pro for T3D. You can read about the IPS Lite on the GitHub page
Or visit my personal website at FuzzyVoidStudio.com

Outline

Visual effects

As you may have noticed, I love particles. So we will work with particles and emitters in this section where I will cover dynamic instancing of objects
What we will create is a small explosion of particles when you pick up a coin.
First we want to define a new datablock for the particle emitter we will be using later.
In *art/datablocks* create a new file **CoinDatablock.cs**
To execute datablocks you should do it in the datablockExec.cs file in the same folder.

CoinDatablock.cs

```
datablock ParticleData(CoinParticle : DefaultParticle)
{
};

datablock ParticleEmitterData(CoinEmitter : DefaultEmitter)
{
   particles = DefaultParticle;
};

datablock ParticleEmitterNodeData(CoinNode : DefaultEmitterNodeData)
{
   timeMultiple = 1.0;
};
```

As you can see we define two new **datablocks** in this file, a **ParticleEmitterDatablock** for creating a new *particle emitter*. *Particle emitters* does not have a place in the world by themselves tho, they only emit particles they need a node to know where to emit particles from. Therefore we need a datablock for a particle emitter node aswell.
What you probably will notice here is when we define the name of the datablocks, we write a colon and then another name. **What does this mean?**
This means that our new datablock, inherits values from another datablock. It makes a copy of the other datablock and lets you edit the values which will only affect the new datablock.
Another thing that is important to note, is that **CoinEmitter** references **CoinParticle**. Which is why it is important that **CoinParticle** is defined *before* **CoinEmitter**

The ParticleEmitter blocks

You have already seen an example of how a datablock is written. But I wrote the CoinParticle for you anyway:

```
datablock ParticleData(CoinParticle : DefaultParticle)
{
    lifetimeMS = 1000;
    gravityCoefficient = 0;
    dragCoefficient = "2";

    sizes[0] = 1;
    sizes[1] = 1;
    sizes[2] = 1;
    sizes[3] = 1;
    inheritedVelFactor = "0";
};
```

You can copy and paste that to the CoinDatablock.cs
Now a little task for you, I want you to create your own emitter. You can either use the ParticleEditor in the World Editor or try different values by writing them in script and then see how it looks in-game.
If you decide to script it in hand, then I wrote a small list of important ParticleEmitter field values.
If you think this is a waste of time, you can find my ParticleEmitterDatablock on the next frame.
However I can ensure you, it is not a waste of time.

My ParticleEmitterDatablock

```
datablock ParticleEmitterData(CoinEmitter : DefaultEmitter)
{
    particles = CoinParticle;
    ejectionPeriodMS = "10";
    ejectionVelocity = "4.167";
    ejectionOffset = "0.625";
    thetaMax = "360";
    softnessDistance = "1";
    lifetimeMS = "200";
};
```

An important thing to note here is that i set the softnessDistance to 1.
It defaults to 1000 so it is important to set this down to something reasonable, or else your particles will look transparent when the background is not kilometres away.

Instantiating new emitters on the fly

Lets put these new datablocks to good use. We want some visual feedback to tell us that we have picked up a coin.
To spawn a new Emitter we will use the **new** operator. It works like this:

```
%emitterNode = new ParticleEmitterNode(){
    datablock = CoinNode;
    emitter = CoinEmitter;
};
```

Remember it is the node not emitter we want to spawn, then we set the emitter inside the "constructor". What i call the constructor is the variable definitions inside the two brackets { and }.
This is where we define what datablock to use, the emitter and anything else we want to do with the newly created object.
We need to give this new object a position in the world.
To get the position of an object you would call **%obj.getPosition();**
And to set the position of an object you would assign it, like this **%obj.position = "x y z";**
Can you figure out where to spawn the new emitter? And how to set it's position (one answer can be found on next frame)
Hint: look in Coin.cs

Spawning emitters when picking up coins

This is how I chose to solve the problem.

```
function Coin::onCollision(%this, %obj, %col, %vec, %len)
{
    %emitterNode = new ParticleEmitterNode(){
        datablock = CoinNode;
        emitter = CoinEmitter;
        position = %obj.getPosition();
    };
    %obj.delete();

    $CoinsFound++;
    if(Coins.getCount() <= 0)
    {
        commandToClient(%col.client,'ShowVictory',$CoinsFound);
    }
}
```

Simply create the new emitter, and give it the same position as the coin.

Schedules and cleanup

If you run into a couple of Coins, and it is all working properly, then if you open the world editor you will notice that the emitters is still there even tho they stopped emitting particles (given that you gave the ParticleEmitter a lifetime) if you didn't you will see that they keep emitting particles. We want to fix that! So I will introduce you to a very important feature in TorqueScript. **Schedules**. You can use a schedule to delete the emitter after some time.
The schedule syntax is:
%obj.schedule(float time, string method, string args...);
Or if you are not calling it on an object:
schedule(float time, int objectID, string method, string args...)
We can use this to delete the emitter after we spawn it:

```
%emitterNode = new ParticleEmitterNode(){
    datablock = CoinNode;
    emitter = CoinEmitter;
    position = %obj.getPosition();
};
%emitterNode.schedule(200,"delete");
```

Gearing the game for multiplayer

So you want to play this brand new game with your friends? Well then there is a few things we need to fix. **Unfortunately** I have deliberately made the game non-multiplayer compatible.
Now lets see how this can be. You remember the **$CoinsFound** global which kept track of the player score?
This global was defined in the **Coins.cs** which resided in the *Scripts/server* directory, and was executed by the scriptExec.cs in the server folder.
The Coins.cs is actually set on the server and the onCollision trigger, increasing the score is also happening on the server. So if there were more than one client they would have the same score!
Maybe that is what you want but that is not what I would want for my awesome coin collecting game. So lets change this code.

The client group, and dynamic variables

Now lets begin, first I will tell you something about dynamic variables. TorqueScript has something called **dynamic variables** which is a very simple but very smart concept.
Basically you can define any variable on any object by assigning it to a value.

```
%obj.somedynVar = 2;
echo(%obj.somedynVar); //outputs "2"
%obj.TSisSpecial = "Is it now?";
echo(%obj.TSisSpecial); //outputs "Is it now?"
echo(%obj.tsisspecial); //outputs "Is it now?"
//(TorqueScript is not case sensitive either.)
```

We can use this to keep track on how many coins each client has picked up!

```
%col.client.coinsfound++; // Automatically starts at 0
```

So what is this ClientGroup I talked about? The ClientGroup is a collection of all the clients who have joined the game. We can use this group to access all the clients and send global messages or iterate through them to find the winner.

Multiplayer support in the coin collection game

Lets start with the simple things. A little task. Remember the **Scripts/client/commands.cs**? We need to edit this file and add a *clientCmdShowDefeat*.
Make this function, use the *clientCmdShowVictory* as a template.
We also need to edit the Coin.cs file. Change the onCollision function to something like the code on the next frame.

```
%emitterNode =  new ParticleEmitterNode(){
    dataBlock = CoinNode;
    emitter = CoinEmitter;
    position = %obj.getPosition();
};
%emitterNode.schedule(200,"delete");
%obj.delete();

%col.client.coinsfound++;
if(Coins.getCount() <= 0)
{
    %winnerClient = %col.client;
    for(%idx = 0; %idx < ClientGroup.getCount(); %idx++)
    {
        %idxClient = ClientGroup.getObject(%idx);
        if(%idxClient.coinsfound > %winnerClient.coinsfound){
         commandToClient(%winnerClient,
              'ShowDefeat',%winnerClient.coinsfound);
           %winnerClient = %idxClient;}
        else
            commandToClient(%idxClient,
                 'ShowDefeat',%idxClient.coinsfound);
    }
    commandToClient(%winnerClient,
         'ShowVictory',%idxClient.coinsfound);
}
```

Networking what happened?

The first thing you should notice is that we swapped the **$CoinsFound** out with *%col.client.coinsfound* so that now the score is on a per-client basis.
Then we changed the wincondition out with a new iterative loop where we find the client with the highest amount of coins found and put him in the first place.
All the clients who did not come in first place will get a defeat message. If someone has higher score than the former first place he will get a defeat message.
Now your first multiplayer game should actually be working! Try opening 2 instances of Torque, in one of the instances you press **"play"** then you tick the **"host"** check box to the left of the **Go** button.
In the other instance you press join, **"Query LAN"** select the server that comes forth and join the game. Now you can compete with yourself about collecting most coins!
Even better, you can host a LAN and let all your friends play your coin collection game with you! Give it a cool name and brag about it a little!

But what can you improve

Use your imagination! I will give you one last task tho, in the above code, there can only be one winner. What happens if two persons have an equal amount of coins collected?
Send any suggestions or needs for improvements to **LukasPJ@FuzzyVoidStudio.com**.
Also if you have a request for a tutorial covering a specific topic please mail me on that address aswell!

Default Datablocks

The DefaultDatablocks we inherit our datablocks is simply some datablocks which is defined in the **Core** folder which allows us for fast prototyping of new emitters.

ParticleEmitterNode important field variables

Point3	alignDirection
	The direction aligned particles should face, only valid if alignParticles is true.
bool	alignParticles
	If true, particles always face along the axis defined by alignDirection.
float	ejectionOffset
	Distance along ejection Z axis from which to eject particles.
float	ejectionVelocity
	Particle ejection velocity.
int	ejectionPeriodMS
	Time (in milliseconds) between each particle ejection.
int	periodVarianceMS
	Variance in ejectionPeriod. Should never be less than 1 or higher than ejectionPeriodMS
int	lifetimeMS
	Lifetime of emitted particles (in milliseconds).
int	lifetimeVariance
	Variance in particle lifetime. Should never be less than 0 or higher than lifetimeMS
string	particles
	List of space or TAB delimited ParticleData datablock names.
float	phiReferenceVel
	Reference angle, from the vertical plane, to eject particles from.
float	phiVariance
	Variance from reference angle, from 0 - 360
float	thetaMin
	Minimum angle, from the horizontal plane, to eject from.

ParticleEmitterNode important field variables (continued)

float	thetaMax
	Maximum angle, from the horizontal plane, to eject from.
float	velocityVarianca
	Variance for ejection velocity, from 0 - ejectionVelocity.
float	softnessDistance
	For soft particles, the distance (in meters) where particles will be faded based on the difference in depth between the particle and the scene geometry.