Arcane FX

02.16.06

user guide

by Jeff Faust

Introduction

Arcane-FX (AFX) is an advanced special effects system for the **Torque Game Engine** (TGE). AFX specializes in spellcasting effects, and with it you can create rich and dramatic magic effects as seen in *World of Warcraft*, *EverQuest II*, and other commercial role-playing games.

The AFX package consists of two main parts: the *AFX Effects Engine* and the *Spellcasting Starter Kit*. The Effects Engine is a programmable framework for choreographing high level thematic effects using existing TGE effects as well as custom AFX effects. The Spellcasting Starter Kit features a collection of example magic spells along with a demo game to showcase the spells in action. The AFX Effects Engine is described in this document, while details about the Spellcasting Starter Kit can be found in a separate document.

Effects Engine

The AFX Effects Engine is the brains of Arcane-FX. As a generalized effects system, it allows many different types of special effects to be controlled using the same method. Explosions, particle emitters, sound effects, animated models, character animations, lights, script events, terrain decals, and more, are all timed and positioned using a common mechanism.

Specifically, the Effects Engine includes a flexible spell system, capable of representing a large and varied assortment of magic spells: quick instant spells, elaborate conjuring spells, long-lasting buff and de-buff spells, damage over time spells, area effect spells, guided projectile spells, creature summoning spells, resurrection spells, teleportation spells, and more.

Component Effects

Component effects are the building blocks of the Effects Engine and are a good place to start learning about AFX. Many of them are effects that you are probably already familiar with. The standard TGE Explosion is one example of a component effect. The AFX custom Zodiac effect is another.

An important consideration of component effects is that they are conceptually separate from the Effects Engine. This is what allows AFX to support pre-

existing TGE effects like `Explosion`, `ParticleEmitter`, and `AudioProfile`. It also means that AFX is easily extended to support custom user effects by implementing a relatively simple adapter class.

Component effects are not limited to visual effects, and include sound effects and script events. Actually, just about anything you want to happen at a specific time and place, can become a component effect under AFX control.

Passive and Influence Effects

Most special effects are *passive* in that they are primarily decorative in nature. Other objects don't collide with them and they don't manipulate or change other simulation objects. Passive effects can usually run exclusively on the clients, effectively invisible to the server. Take particle emitters as an example. You may have identical emitters running on different clients, but it's usually not important that each particle they generate matches between them.

There is another category of effect called an *influence* effect. These effects do something to other simulation objects and therefore usually have to execute on the server, or both the server and clients. Examples of influence effects include `afxCameraShake` and `afxAnimClip`. `afxCameraShake` causes the camera or other objects to move around, while `afxAnimClip` selects animation sequences in `Player` objects.

Effect Wrappers

Effect Wrappers are what make it possible for the Effects Engine to make use of special effects outside of the system. An important assumption made in the Effects Engine is that it does not need to know much about specific component effects in order to combine them into high level thematic effects. To the engine, most effects are just black boxes that need to be told when and where to do whatever it is that they do.

As the name implies, Effect Wrappers wrap around or enclose existing effects and make them all look the same to the rest of the engine. In object oriented programming terms, they provides sort of an after-the-fact polymorphism between effect types.

For each component effect used in an AFX effect, a corresponding effect wrapper is created using an `afxEffectWrapperData` datablock.

`afxEffectWrapperData` knows about a large number of different component effects and it can be extended to recognize additional component effects with the addition of a small amount of customized code.

TGE Component Effects

Following is a summary of standard TGE effects currently recognized by the AFX Effects Engine. Except where noted, these effects are unmodified from their form in TGE. In all cases, they should be compatible with any pre-existing datablocks you may already have defined for these effect types.

AudioProfile – *passive effect*

AFX supports audio effects using standard TGE `AudioProfile` datablocks which in turn utilize `AudioDescription` datablocks. Audio effects are created from `AudioProfiles` using `alxPlay()` and managed internally. AFX dynamically controls an audio effect's location and when it shuts off possibly with a fade out. Both looping and one-shot sound effects are supported.

CameraShake – *influence effect*

AFX utilizes the standard TGE `CameraShake` class to implement a camera shake effect. `CameraShake` is not normally specified with a datablock, so this is done using `afxCameraShakeData` datablocks. The TGE `CameraFXManager` is not used. Instead, the `CameraShake` class is managed by the Effects Engine. In spite of its name, this effect is not limited to shaking the camera. The target of its effect is determined by the AFX constraint system.

Debris – *passive effect*

The `Debris` effect is a standard TGE effect which can also be used as an AFX component effect. AFX uses `DebrisData` datablocks along with the `Debris` class to create client-only debris. AFX controls when and where `Debris` is used including its initial direction vector, but once the debris is triggered it must resolve on its own. Once started, AFX won't interrupt the debris early or dynamically change anything about it.

Explosion – *passive effect*

The `Explosion` effect is a standard TGE effect which can also be used as an AFX component effect. AFX uses `ExplosionData` datablocks along with the `Explosion` class to create client-only explosions. AFX controls when and where an explosion occurs, but once the explosion is triggered it must resolve on its own. Once started, AFX won't interrupt the explosion early or dynamically change anything about it.

ParticleEmitter – *passive effect*

The `ParticleEmitter` is a standard TGE effect which can also be used as an AFX component effect. AFX uses `ParticleEmitterData` datablocks along with the `ParticleEmitter` class to create client-only particle emitters. AFX dynamically controls a particle emitter's location, orientation, and when it is actively emitting particles. AFX actually uses an enhanced `ParticleEmitter` for some spells in the Spellcasting Starter Kit, but the standard `ParticleEmitter` can be used if the enhancements are not needed.

AFX Custom Component Effects

afxAnimClip – *influence effect*

An `afxAnimClip` forces a target `Player` or `aiPlayer` object to play a particular animation sequence. The `afxAnimClip` does not supply any animation data, it merely selects, by name, a sequence that is already defined in the target. Normally when an `afxAnimClip` is applied to a user-controlled `Player` object, any interactive user actions will override the animation selected by the `afxAnimClip`, but `AfxAnimClips` can be configured to temporarily block out user actions.

afxAnimLock – *influence effect*

Although `afxAnimClips` can be configured to block out user actions, sometimes it's only appropriate for the user to be blocked out for a short section of a longer playing `afxAnimClip` animation. `afxAnimLock` can be used to set a specific timespan when user actions are blocked, independent of `afxAnimClip` timing.

afxLight – *passive effect*

Dynamic lighting in TGE is supported at a low level, generally allowing any subclass of `SimObject` to act as a light using the `registerLights()` method. AFX adds the simple `afxLightData` and `afxLight` classes for implementing client-only dynamic light effects. `afxLight` supports all of the TGE lighting types: `Point`, `Spot`, `Vector`, and `Ambient`, but in practice, `Point` is the most useful because the other types have no effect on terrain and interiors.

afxModel – *passive effect*

TGE has a number of model classes, but lacks an effective client-only model effect. AFX fills this gap with the lightweight `afxModelData` and `afxModel` classes. `afxModel` loads a single dts format model and can play one animation sequence stored in the model. It also has settings for transparency and can override some material settings.

afxScriptEvent – *script effect*

Arbitrary script functions can be called as an AFX effect using `afxScriptEvent`. It is neither a passive or influence effect in that it depends what the script does. Only server `afxScriptEvents` are supported.

afxZodiac – *passive effect*

AFX includes a very useful decal-like effect called an `afxZodiac`. `afxZodiacs` are decal textures that are always projected vertically onto the ground and are often circular. Parameters control dynamic rotation, and scale as well as texture, color, and blending style. `afxZodiacs` are very effective as rune lighting rings that rotate below a magic user while casting a spell. They are also useful for explosion shockwaves and scorched earth decals. `afxZodiacs` work on both terrain and interiors.

Effect Choreographers

In AFX, high level special effects are created and coordinated using Effect Choreographers. At this time, the only implemented Effect Choreographer is

the Magic Spell Choreographer, but a number of different choreographers are planned.

Magic Spell Choreographer

The `afxMagicSpell` choreographer is for creating spellcasting effects and it was the first choreographer implemented in AFX. `afxMagicSpell` manages the concept of a magic spell from the time spellcasting begins until the spell finally dissipates.

Spell Structure

Since magic spells are a fantasy construct and there are no real world examples to study, the nature of a spell's behavior is open to interpretation. `afxMagicSpell` attempts to reproduce the magic behavior found in many computer role-playing games.

Specifically, an `afxMagicSpell` has up to 5 distinct phases: *Casting*, *Launch*, *Delivery*, *Impact*, and *Linger*. Casting, Delivery, and Linger often cover a span of time, while Launch and Impact are always a specific moment in time. An `afxMagicSpell` passes through each of these phases in order, but often one or more of the Casting, Delivery, and Linger phases are reduced to a zero duration and effectively ignored.

Casting -- The Casting phase is a warmup period in which the spellcaster prepares to cast a spell. Typically, this is treated as a time period where the spellcasting can be interrupted by damage to the spellcaster, or if the spellcaster moves. The length of casting time is an important element for game balancing, as it influences the vulnerability of the spellcaster and affects how often the spell can be repeated. Some spells, often called *instants*, have a zero duration casting time and proceed immediately to the Launch event.

Launch -- The Launch is the specific moment when the spellcasting is completed, and the spell now exists independently of the spellcaster. This is usually the time when the spellcaster pays the *mana* cost of casting the spell.

Delivery -- Delivery phase is an open-ended period of time where the spell moves from the spellcaster to the location of a target. Often Delivery is determined by a projectile that moves through space and may or may not hit its intended target.

Impact -- Impact is a specific moment when the spell reaches a target or possible when the delivery phase exceeds a time limit.

Linger -- The Linger phase is a period of time where the spell continues to affect a target.

Spell Effects

Spell effects are organized around the five phases: Casting, Launch, Delivery, Impact, and Lingering. For each phase, there is an independent list of effects connected to it. Each phase establishes a localized time span or instant off of which the timing of its effects are based. For example, take an effect that is configured to start at a time of 1 second and last for 2 seconds. If that effect is attached to the Casting phase, it will start 1 second after the Casting phase starts and last for 2 seconds. While effects cannot start before the start of the phase they are attached, they can last beyond the end of the phase and overlap effects in the next phase.

Constraint System

When designing special effects, one of the best sources for animation is other animated objects. The AFX constraint system allows you to directly “borrow” the animation of other objects and use it to place your effects. Each effect wrapper specifies three separate constraints. A *position constraint* which specifies a 3D location, an *orientation constraint* that specifies an orientation or bearing, and an *aim constraint*, which is another positional constraint, usually used to point the effect towards. (Several other constraints are currently being considered: an *influence target* for indicating what object an influence effect should act on, a *condition constraint* indicating an object that can be tested for conditional effect execution, and a *timing constraint* that would contain timing information.)

Constraint Objects

The constraint system supports several different types of constraint object, point constraints, object constraints, and shape constraint. A point constraint is a point is space. Spell effects that are active after the spell's missile impacts a target, can be constrained to the impact position, which is a point constraint. An object constraint allows constraining to the origin of a scene object. Effects can be constrained to a spell's missile projectile which is an object constraint.

A shape constraint allows constraining to the origin of a shape or to one of the shape's hierarchical nodes. The spellcaster character is a shape constraint and effects can be constrained the the spellcaster's origin, or to one of its nodes such as a hand or a foot.

Constraint History

A powerful feature of the constraint system is constraint history. Constraint objects can be configured to record a short history of past positions and/or orientations. Both the length of history recorded and the sampling rate can be configured. Effects constrained to a constraint with history can refer to old values of the constraint within the recorded range.

Constraint history is useful for creating effects that follow behind another object. Constraint history is used in the Spellcaster Starter Kit spell called *Insectoplasm,* where model segments line up behind an erratically moving missile to form a giant centipede-like monster.

Transform Modifiers

While the AFX constraint system is very powerful, all you can really do with it by itself is hang your effects onto your characters and objects like ornaments on a Christmas tree. Usually, you want something a bit more dynamic than that, and in AFX you get more by using *transform modifiers*. A transform modifier is a code module that changes an effect's current location, orientation, and/or aim location. One transform modifier might move an effect's position by a simple offset. Then another might rotate the effect about an arbitrary axis. Yet another, adjusts the effect's position vertically to a fixed height above the ground. By arranging transform modifiers into a sequence of operations, you can achieve a large variety of procedural animation behaviors.

Transform modifiers are used in effect wrappers and are arranged as a sequential list of modifiers. Each modifier picks up where the last modifier left off and further modifies an effect's location, orientation, and/or aim location. For example, use a small LocalOffset followed by a Spin and your effect will spin on its own axis a short distance from its constraint position. Reverse the order, placing the Spin first, followed by the LocalOffset, and your effect will orbit its constraint position.

afxXM_WorldOffset

A 3D offset is simply added to the effect's current location.

afxXM_LocalOffset

A 3D offset is reoriented using the effect's current orientation and then added to the effect's current location.

afxXM_Spin

The effect's current orientation is modified to rotate around a specified axis, at a specified rate and starting angle.

afxXM_GroundConform

The effect's current height, (the Z component of its position), is adjusted to a specified height above the terrain or interior below it.

afxXM_Aim

The effect's current orientation is adjusted to face its positive Y axis toward the effect's aim constraint.

afxXM_PathConform

The effect's current position is adjusted to conform to a specified path. It's orientation may also be adjusted to orient to the path.

afxXM_Freeze

An effect's position, orientation, and/or aim constraint is sampled exactly once, and then repeated from then on, regardless of changes in the constraints.