Loading in Away3D 4.0 (Broomstick)

May 4th, 2011, By Richard Olsson (r@richardolsson.se)

This document outlines the structure of the new loading framework and asset library in Away3D 4.0 (codename "Broomstick".) At the time of writing, Away3D 4.0 is still alpha and the loading framework has just been pushed to the master branch in Git. Please regard this document as a top-level overview of the functionality. Things can still change meaning that the code snippets might not work completely, but the overall idea should still remain the same.

Questions should be directed to the away3d-dev mailing list on Google Groups, http://groups.google.com/group/away3d-dev.

The Away3D 3.x approach to loading

In earlier versions of Away3D, there were essentially two very similar ways of loading a file. The first way was to use an instance of the Loader3D class directly, which would let you initiate a load and then instantly add the Loader3D instance to the scene, a lot like the native Flash Loader class works. You had to specify which file format to use by instantiating a parser class and pass it to the load() method.

The other way was to use the static load() method on the parser class. This would still create a Loader3D to load the file, and return that Loader3D instance, so was essentially the same thing but using an only slightly different interface.

This was not a very versatile solution, providing only very basic functionality. Some major drawbacks were that the contents of a loaded file had to be added to the scene (as part of the Loader3D container) or there was no way to retrieve them after a load had finished, and that there was no way of knowing exactly what had been loaded other than looping over the children of the Loader3D.

Design goals for loading in Away3D 4.0

When designing the new loading framework in Away3D 4.0, we have gathered our collective experience about loading and managing assets in earlier versions of Away3D, and defined a set of goals that we want the new framework to meet. The most important of these are:

- The framework needs to work with any type of file format that is likely to be used in a 3D application. Many file formats contain data that is not simple scene data, and some don't contain scene data at all (e.g. only animations.) The framework need to be able to load these and make them available.
- Promote good practice. Loading is an asynchronous operation, yet the old Loader3D discouraged the programmer from caring about asynchronous events since it could simply be added straight to the scene (and was the only class available for loading.)
- Aid runtime asset management. Based on experience from building Prefab3D it was concluded that a central library where all assets end up and can be retrieved at any point during runtime was a necessity for complex apps.
- Be extendable. In the event that parts of the Away3D 4.0 loading framework does not fit a particular user's application, that user should be able to cherry-pick the parts of the framework that do fit, and build new systems around those.
- Provide some sort of synchronous/proxy solution. Despite many shortcomings, the old

Loader3D approach was great in certain cases because it made loading of simple scene data and displaying it a very simple task. Something equivalent needs to exist in Away3D 4.0 as well.

- Be ready for the Molehill era. We are now dealing with much more complex data and more importantly, much *more* data than ever. Parsing such complex data must perform well and be possible without locking the process, e.g. via "green-threading".

With these goals in mind, we designed a much more competent asset loading and management system than we have ever had in Away3D. This is described over the next few pages.

The Away3D 4.0 Asset Library

The biggest addition to Away3D 4.0 in terms of asset management is the AssetLibrary class, which acts as a central repository (or several if you want) for assets that you load into Away3D. The purpose is to let you load assets at one point (e.g. in a pre-loader module) and then access and use those assets at any other point in the application.

Using the library is very simple. The AssetLibrary class provides a static interface to load a resource (e.g. a file) and retrieve assets (e.g. models, textures, animations) that have already been loaded as part of a resource. A simple example looks like this:

```
// To load
AssetLibrary.load(new URLRequest('path/to/my/file.awd'));

// Once load has finished
var mesh : Mesh = Mesh(AssetLibrary.getAsset('nameOfMyMesh'));
```

If you prefer to store an Asset Library instance and pass it around, it can also be used via a singleton-like interface. The following two lines of code are completely equivalent.

```
AssetLibrary.getAsset('name');
AssetLibrary.getInstance().getAsset('name');
```

In the rare case that you need to have several asset libraries (e.g. when writing a modular application where the modules should not be sharing assets) the singleton interface is actually a "multiton", meaning you can get different instances by ID. The following two lines reference assets by the same name but from two different asset libraries.

```
AssetLibrary.getInstance('main').getAsset('myAsset')
AssetLibrary.getInstance('module').getAsset('myAsset');
```

The default instance, used internally when using the "singleton" or static interfaces, is called "default".

Assets can be used either once the entire resource has been loaded, but also once the single asset has been encountered within that resource. To detect whenever an asset is encountered in a file, listen for the AssetEvent.ASSET_COMPLETE event which will contain a reference to the asset.

Simple loading without the AssetLibrary

For those cases where no library management is necessary (or when you want to create your own asset management system) the AssetLoader can be used directly. The AssetLibrary uses AssetLoader internally, and the interface is more or less identical.

```
var loader : AssetLoader;
loader = new AssetLoader();
loader.addEventListener(LoaderEvent.RESOURCE_COMPLETE, onComplete);
loader.load(new URLRequest('path/to/my/file.awd'));
```

As with the AssetLibrary, the ASSET_COMPLETE event will be dispatched whenever an asset is encountered in the file (or it's dependencies.)

AssetLoader (and hence AssetLibrary) is able to retrieve dependencies (e.g. external textures referenced in a file) automatically as part of a single resource. In fact, in most cases, provided that the URLs are correct, the dependency loading will be completely transparent and just work.

If you want to build your own dependency management, use the SingleFileLoader class directly, which is otherwise used for each dependency internally in AssetLoader.

Using Loader3D for simple scenes and models

The Loader3D still exists in Away3D 4.0, but should now only be used for cases where it actually works well, that is with simple models and scene files. The Loader3D uses either AssetLibrary or AssetLoader internally (toggled using a constructor parameter) which means that you get all the benefits of these classes, while at the same time being able to add the Loader3D directly to the scene without having to wait for the loading to finish. Scene objects that are encountered during loading (like meshes, containers and lights) will be added to the Loader3D container, whereas all other assets will be ignored (albeit still saved in the library if using AssetLibrary for loading.)

```
// Use AssetLibrary with ID "main"
var loader : Loader3D = new Loader3D(true, "main");
loader.load(new URLRequest('path/to/my/file.awd'));
myScene.addChild(loader);
```

Scene objects will start appearing as the file is loaded and parsed. Any events dispatched by the internal AssetLoader/AssetLibrary will be bubbled by the Loader3D.

File format auto-detection and plug-in parsers

You may have noticed that none of the examples so far pass a parser instance to the load() methods the way that was necessary in earlier versions of Away3D. This is another big new feature in Away3D 4.0; automatic detection of file formats. This means that you can now load a file without knowing it's file format (provided that it is one supported by Away3D) and instead use the exact same load() call (with different URLs) for all you loading operations.

The parsers themselves are responsible for indicating whether they support a file or not and there is no special code for this internally in the loaders. This means that you can write and plug in your own parsers into this system if you want to. As a load initiates, all parsers will be queried for the suffix in the URL (e.g. .3ds or .obj) and will indicate whether or not they recognize the file format. In case no suffix is available or none of the parsers recognizes it, a second round of checks will be performed on the actual data once it's loaded. This means that the system still works even with embedded data or with assets loaded from URLs like assets.php?assetId=123.

This system requires the loader classes to have references to the parsers, and hence that the parsers are compiled into the SWF. This can potentially add more than 100kb to a SWF that would otherwise be just 40kb (i.e. banner size) and to prevent this from happening in every single case, we have opted not to include parsers by default. This means that you are required to plug parsers into Away3D at a single point in your application, e.g. during initialization (where you usually create your View3D.) You do this using the enableParser() and enableParsers() methods on any of the loading classes.

```
Loader3D.enableParser(OBJParser);
Loader3D.enableParser(AWDParser);
```

If you don't care about file size, the constant ALL_BUNDLED on the Parsers class can be used to conveniently add all bundled parsers at once, making this a single-liner:

```
Loader3D.enableParsers(Parsers.ALL_BUNDLED);
```

These static methods exist on all the loading classes and add the parsers to the same internal repository. Hence, there is no need to add the same parser both to the Loader3D and the AssetLibrary (it will be added to the same place and ignored the second time it's added.)

If you do not want to use this system, you can still create your own parser instances and pass to all of the loading methods on Loader3D, AssetLoader and AssetLibrary, the same way that you would in earlier versions of Away3D.

```
var parser : OBJParser = new OBJParser();
AssetLoader.load(new URLRequest('myfile.obj'), parser);
```

If you haven't enabled any parsers (using enableParser() or enableParsers()) and still do not pass a parser instance when loading, an error will occur since the loaded data cannot be parsed.

Configuring the loading operation

The third parameter of all loading methods is an optional AssetLoaderContext instance. Instances of this class are used to configure a single loading operation. The main use is to define whether dependencies should be loaded automatically, and where the AssetLoader should look for those dependencies. To disable dependency loading altogether, which is enabled by default, set the includeDependencies property (or constructor parameter) to false.

```
var context : AssetLoaderContext = new AssetLoaderContext(false);
var loader : Loader3D = new Loader3D();
loader.load(new URLRequest('file.awd'), new AWDParser(), context);
```

To configure a URL prefix for all dependencies, set the dependencyBaseUrl property on the context instance.

```
context = new AssetLoaderContext(true, '/path/to/dependencies');
```

Unless the dependency URLs (in the file) are absolute, using the above context will prepend /path/to/dependencies to all dependency URLs. Absolute URLs (e.g. those that begin with http:// or equivalent) will be used as is without the prefix.

Embedding files and dependencies

If you prefer to embed data directly into your SWF files (using the [Embed] ActionScript meta tag) you can still use all of the features described above. All of the loading classes have a parseData() method which lets you pass any type of data, which will then be parsed as if it had been loaded, provided that there is a parser that supports and recognizes it. You can even pass a Class, which is what you get when you embed a file in AS3/Flex, making it a very simple piece of code:

```
[Embed(source="/path/to/file.awd", mimeType="application/octet-stream")]
private var FileDataClass : Class;

Loader3D.enableParsers(Parsers.ALL_BUNDLED);

var loader : Loader3D = new Loader3D();
loader.parseData(FileDataClass);
```

```
myScene.addChild(loader);
```

In this case, the file will be recognized as an AWD file (since the AWDParser has implicitly been enabled as one of the bundled parsers) and the data in the file will be parsed for assets. As scene assets are encountered they are added to the Loader3D container as usual, making them appear on screen.

If the embedded file references dependencies, Away3D will attempt to load them at the URL by which they are referenced in the master file (the one called "file.awd" in the above example.) When embedding files, chances are that you do not want this to happen, but rather have the dependencies embedded as well. For these cases the AssetLoaderContext provides a way to map dependency URLs to embedded data, using the mapUrlToData() method on context instances.

```
[Embed(source='path/to/texture.jpg', mimeType="application/octet-stream")]
private var EmbeddedImage : Class;

var context : AssetLoaderContext = new AssetLoaderContext();
context.mapUrlToData('path/to/texture.jpg', EmbeddedImage);
```

Using the context created above in a load() or parseData() call will resolve any reference to "path/to/texture.jpg" from the loaded file to the EmbeddedImage data. Note that the image needs to be embedded with MIME-type application/octet-stream for it to be a ByteArray, or Flex will automatically embed it as a BitmapDataAsset, which will not work.

You can combine external loading with embedding in any way, using the load() and parseData() methods along with the AssetLoaderContext. For instance, you can load files but embed their dependencies, or vice versa, or embed only certain dependencies.

Summary

In Away3D 4.0 there are three main ways to load data. Using the AssetLibrary class, for both loading and management of assets. Using the AssetLoader directly to circumvent the management system, but still be able to conveniently load files with many dependencies. And finally, the Loader3D class continues to exist in Away3D 4 to provide a quick and dirty way to load visible assets into a scene.

The AssetLibrary can be used via it's static interface, or using singleton or multiton patterns.

Parsers can either be passed manually to the load() and parseData() methods, or selected by the automatic file format detection provided that parser classes have been plugged in using the enableParser() set of methods.

Both external files and embedded data can be used equally with all loading classes, using the load() and parseData() methods respectively on each of the loader classes.

Use instances of the AssetLoaderContext class to configure a loading operation and how to deal with dependencies.