NLS GAME ENGINE SCRIPTING INTERFACE DESIGN DOCUMENT

Consider this an RFQ - comment freely. I'm still in the design / working on the implementation phase - hence while the below may be written in the current tense, as if the engine currently did what was stated, it might not yet be created/working that way yet. ~Ricky

Table of Contents

Revisions

2012-10-02 - Ricky

- Added SYSTEM_DIRS::TEMP to allow access to the system-wide temporary folder.
- Removed the TYPE specifier from all enumerations as that workaround is no longer needed in AngelScript.

2012-07-01 - Ricky

- Changed the Rotation::Slerp method to be a modifying operation and added Rotation::SlerpCopy to give the previous behavior. This matches much more closely with the Vector methods.
- Added the AngleTo method to provide a way of getting the angle between the current rotation and a given rotation.

2012-05-28 - Ricky

- Noted that the Rotation types allow non-unit operations.
- Added a note about the flavors of Rotation notation.

2012-05-27 - Ricky

- Clarified the methods of the Vector types that modify the current vector.
- Added the *Copy methods to the Vector types for those times when scripters explicitly want the original to remain unchanged.
- Added the ApplyRotation* methods to the Vector types to provide an alternate syntax for those who dislike the operator overloads.

2012-05-21 - Ricky

- Added more operations to the Vector type.
- Added aliases to the various Vector, Rotation, &c types.
- Renamed Envelope to GenericArray and PropertyMap to GenericMap to reflect the true purpose and meaning of these types: they are generic containers able to hold any type you want to place in them in any mixture you want.

2012-04-09 - Ricky

- Removed the Scene system I started. After more thought I realized that different modules will need different ways of looking at the scene architecture; most physics engines have their own hierarchy systems, as do most rendering engines. Thus hierarchies of Entities are module-specific and cannot be handled in a generic manner by the global engine.
- After conversation with Adam, I added some flags to help detect the status of a module and a function to reset the module.

2012-04-08 - Ricky

- Added revisions to the document so that the decisions can be exposed, and readers can know when something has changed.
- Switched to AS 2.23 to gain API namespace support. This allows for freedom to change

the API while still providing API stability to those that need it.

- Removed the Engine object in favor of just exposing the member methods as functions under the new Engine API.
- Cleaned up the enumeration listing by cutting out the visual redundancy. Users will quickly learn that they must put in the namespace to access enumeration members. Example: namespace ::Engine { LOG(LOG_PRIORITY::INFO, “Hello”); }
- Removed the message and update handler registrations as a general-purpose message router is not needed. Instead an engine-specific callback system has been designed. Modules that need to communicate events can do similar.
- Fleshed out Envelope and PropertyMap.
- Added a Shutdown function to replace the old CORE_MESSAGE::QUIT.
- Documented the new route for creating entities and attaching components to them in the Design notes.
- Created a section documenting the differences between the pointer and POD types - and how they are used.
- Switched out EntityList for a more advanced Scene management concept.

2012-04-06 - Ricky

- Started organizing the loose thoughts into a more documentation approach.
- Changed the config flags to a more sane method; two different flags for the same task didn’t make sense. Better to just make it a single flag with a boolean.
- Added a new flag for setting/overwriting the configuration log file location. This cannot be predetermined in code as the location is platform and configuration dependent. Therefore it is better to let it default to the executable directory in Debug, as it’s quite likely the user will have write privs on that folder in that situation, but in Release it is highly unlikely that they will have such privs. In the last case if such a log is needed, or in the former case if the user/developer wants the log somewhere specific, it can be set via the commandline flag.
- Started on the bulk of the types, enums, objects, functions, etc. that should be available to each script type.

2012-04-05 - Ricky

- Broke scripting into two parts, config and game, to help rid the engine of the hacky NLSSD config file and to make configuration more dynamic.
- Added an initial round of configuration switches to help choose between precompiled and uncompiled scripts.

Design notes

Since the modules, once loaded and the engine switches over to the game scripts, add module-specific commands to the script language, it's not possible to list everything that the engine can do. That said, there are some common things the engine provides, and those are listed here.

The below now assumes AngelScript 2.23 minimum to provide namespace registration. Every module should be registering its interface under a module-specific, maybe even module-version-specific, namespace. All the below indicate whether they are in the global namespace "::Item" or in the Engine's namespace(s). Going into the future, as the script interface evolves, any backwards compatibility breaking changes will increment the minor revision number of the interface and then be introduced into the main engine namespace and into a new version-specific namespace, but the older interface will be marked deprecated and retained for at least one version cycle in its version-specific namespace. This means new compatibility-breaking features and changes will arrive in "::Engine::" as well as "::Engine-X.Y::" while the older "::Engine.A.B::" will retain all the old interfaces. If a change is deemed to be backwards compatible, then no new version or namespace will be added. Old namespaces will be dropped every major version number of the engine, just to help keep the engine clean.

Because of the namespacing, it can be difficult for script writers, due to the added verbosity. To mitigate this it is recommended to wrap scripts in the engine namespace that matches the version used. This will protect the scripts from compatibility breaking API changes, as well as explicitly document the API version the script uses.

Older users of the engine may wonder where the functionality is located for connecting components to Entities. Put it this way: CreateEntityFactory is now a petrified dinosaur that is long dead and buried - may the festivities not cease. In its place modules will directly expose functions/methods for creating components. Each of these functions will require Entities to be passed to them so that they know what that component is supposed to be attached to. They may even, depending on the component being created, require more information. This should simplify and clarify entity creation immensely!

Scripting-related command-line switches

- --asc 0 Engine attempts to look for raw Angelscript source code (default in Debug)
- --asc 1 Engine attempts to look for precompiled Angelscript files (default in Release)
    - In either case, if the requested type isn't found, the engine will report the fault to

either the GUI or the log. The location of the report depends entirely on when the error happened.

- --config-log *file* Writes the log generated by the engine during the config stage to the specified file. If the file cannot be created/modified the engine will display a GUI error. In Debug this defaults to the same folder as the executable, while in Release this log is not generated unless specified via this switch.

Basic types vs pointer types

In the scripting interface there are two different types, basic/POD types and pointer types. Unlike C++, the syntax is largely the same for both. That said, there are still slight changes of use between them. One of the most critical is creation:

```
Entity ent; // As Entity is a pointer type, this is an uninitialized pointer.
Vector vec; // As Vector is a POD type, this is a zero vector.
```

As you can see, if you don't call a constructor for the pointer types the code may crash in rather strange ways. While having loose pointer variables around has its uses, the safer way to create pointer types is by calling one of its constructors - which can be as simple as appending a pair of parentheses:

```
Envelope envelope(); // Creates a new, empty envelope.
Entity ent("Rock31"); // Creates a new named entity.
```

Two script types

Config script

While the engine as previously used a, rather clunky, config file to load up the game, it now uses a config script. This is different from the game script(s), as it has access to very little - namely the configuration options of the engine, such as:

- Listing and loading modules
- Setting the per-user data path
- Setting the log file name
- Loading per-user config {scripts|files - pick one}
- Setting asset paths - including where the game scripts are located
- Setting initial game script name and the function to call
- Detecting operating system and GUI

Enumerations

- ::Engine::SYSTEM_DIRS - all available directory locations
    - USER The user's home folder

    - DOCUMENTS The user’s documents folder
    - PICTURES The user’s pictures folder
    - MUSIC The user’s music folder
    - VIDEO The user’s video folder
    - DESKTOP The user’s desktop folder
- ::Engine::MODULE_STATUS - the status of a module
    - NOT_FOUND Library does not exist.
    - LOAD_ERROR Library loading error.
    - START_ERROR Module starting error.
    - EXISTS Library exists and is available for loading.
    - LOADED Library loaded and module started.
- ::Engine::LOG_PRIORITY
    - INFO General information.
    - FLOW Program flow for debugging purposes.
    - WARN General warnings, usually about data that is not correctly formed, but wasn't a showstopper.
    - CONFIG Errors with the configuration - usually showstoppers, but not always; the code might be able to continue using a default value.
    - ERR Errors in operation - usually showstoppers, but not always.
    - MISSRESS A requested resource doesn't exist: missing files, etc. A default may be provided.
    - DEPRECATE Deprecation warning.
- ::Engine::ASSET_LOCATION - The location of game assets, all stored relative to the engine executable.
    - SCRIPT The location of game scripts.
    - SOUND
    - TEXTURE
    - GEOMETRY
- ::Engine::OPERATING_SYSTEM - the various supported operating systems
    - WINDOWS
    - APPLE *{unsupported}*
    - APPLE_IOS *{unsupported}*
    - ANDROID *{unsupported}*
    - LINUX *{unsupported}*
- ::Engine::USER_INTERFACE_API - the various supported GUI APIs
    - WINDOWS
    - APPLE *{unsupported}*
    - APPLE_IOS *{unsupported}*
    - ANDROID *{unsupported}*

    - QT *{unsupported}*
    - GTK *{unsupported}*
    - WXWIDGETS *{unsupported}*

Functions

- Logging
    - **::Engine::LOG**(LOG_PRIORITY, string message); // Send a message to the log file
        - Script API as above
        - Implemented and tested (not unit tested though).
- Parameter settings
    - **::Engine::SetUserDataFolder**(string); // Sets the folder, relative to the user's application data folder, where the engine and game should store per-user information. Example: ::Engine::SetUserDataFolder("/Publisher/Game");
    - **::Engine::SetLogFile**(string name, string file_extension); // Sets the file name (including file extension) of the log. The log will be placed in the per-user data path. Up to 5 backups of previous logs will be kept. Not until this AND SetUserDataPath have both been set will the log file be written to disk, so it's imperative that these get set near the top of your script. (See command line switch --config-log) Example: ::Engine::SetLogFile("game", "log");
    - **::Engine::SetAssetFolder**(ASSET_LOCATION, string);
    - **::Engine::SetGameScript**(string file, string function); // Sets the game script file name, excluding file extension, and the declaration of the method to call. Example: ::Engine::SetGameScript("main", "void main(void)");
- Informational queries
    - MODULE_STATUS **::Engine::GetModuleStatus**(string);
    - OPERATING_SYSTEM **::Engine::GetOS**();
    - string **::Engine::GetOSVersion**();
    - USER_INTERFACE_API **::Engine::GetGUI**();
    - string **::Engine::GetFolder**(SYSTEM_DIRS); // Get a known folder path.
- Module loading
    - array<string> **::Engine::ListAvailableModules**(); // Returns an array of all the module names that can be loaded.
    - MODULE_STATUS **::Engine::LoadModule**(string);
        - Script API as follows: **Engine::ModuleLoader.LoadModule(string);**
        - Implemented and tested (not unit tested though).
- Engine state
    - **::Engine::Shutdown()**; // Instructs the engine to stop all activity and quit.
- Per-user config loading

- *{TODO: methods for per-user config handling}*

Game script(s)

Game scripts are where all the game logic is stored. While some engine parameters are tweakable from here, the primary purpose of the game scripts are to house the game's logic. Here it is a lot more difficult to list what is available to the scripts, as each module provides something different. However, there are some basic tools provided by the engine that won't change.

Enumerations

- ::Engine::EVENT
  - UPDATE This event allows the script to do animations and other tasks that need regular updates. As this event is called 30 times a second or so, be sure to keep the longest code path in the handler fairly short in order to not slow things down! Handlers are required to have the following signature: void (double)
  - SCRIPT_ERROR This event is called whenever there's a non-critical error in the script, allowing the script to detect the fault and take action. Critical errors, aka script compilation errors, cause the engine to abort immediately. Handlers are required to have the following signature: void (string, string, uint, uint)
- ::Engine::LOG_PRIORITY
  - INFO General information.
  - FLOW Program flow for debugging purposes.
  - WARN General warnings, usually about data that is not correctly formed, but wasn't a showstopper.
  - CONFIG Errors with the configuration - usually showstoppers, but not always; the code might be able to continue using a default value.
  - ERR Errors in operation - usually showstoppers, but not always.
  - MISSRESS A requested resource doesn't exist: missing files, etc. A default may be provided.
  - DEPRECATE Deprecation warning.
- ::Engine::MODULE_STATUS - the status of a module
  - NOT_FOUND
  - NOT_LOADED
  - RUNNING
  - ICY The engine has detected multiple recent freezes, but the module has recovered each time. The module may be overloaded.
  - FROZEN The engine hasn't received a watchdog clear within 100ms.
  - CRASHED The engine detected that the module has stopped working.

- ::Engine::SYSTEM_DIRS - all available directory locations
    - USER The user's home folder
    - DOCUMENTS The user's documents folder
    - PICTURES The user's pictures folder
    - MUSIC The user's music folder
    - VIDEO The user's video folder
    - DESKTOP The user's desktop folder
    - TEMP The system's temporary folder

Data Types

- ::Engine::Vector - a 3D floating-point vector. Comes in several flavors, but the below only documents the Vector3 which is also given the name of "Vector" - so you can either just run with calling one a Vector, or you can be explicit and call it a Vector3.
    - Flavors:
        - Vector2 - a two-element floating-point vector.
        - Vector or Vector3 - a three-element floating-point vector. Note that these two are implicitly castable, as they are considered the same type.
        - Vector4 - a four-element floating-point vector.
        - Vector2i - a two-element integer-based vector.
        - Vector3i - a three-element integer-based vector.
        - more available if needed...
    - Constructors:
        - Vector myVector(); // A zero vector.
        - Vector myVector(float x, float y, float z); // A vector with the given parameters.
        - Vector myVector(Vector); // Performs a deep copy of the passed vector.
    - Properties: **x, y, z**
    - Methods:
        - float **MagnitudeSq**(); // Returns the squared magnitude of the vector. Is a LOT faster than the Magnitude method.
        - float **Magnitude**(); // Returns the magnitude of the vector.
        - float **DistanceSq**(Vector other); // Returns the squared vector distance between this vector and the other.
        - float **Distance**(Vector other); // Returns the vector distance between this vector and the other.
        - void **Normalize**(); // Normalizes this vector.
        - Vector **NormalizedCopy**(); // Returns a new vector that is the normalization of this vector.
        - float **Dot**(Vector other); // Returns the dot product of this vector and the

passed vector. Equivalent to {vec * other}
        - void **Cross**(Vector other); // Applies the cross product of this vector and the passed vector to this vector.
        - Vector **CrossCopy**(Vector other); // Returns a new vector that is the cross product of this vector and the passed vector. Equivalent to {vec % rot}
        - void **ApplyRotation**(Rotation rot); // Applies the rotation to this vector.
        - void **ApplyRotationInv**(Rotation rot); // Applies the inverse of the rotation to this vector.
        - Vector **ApplyRotationCopy**(Rotation rot); // Returns a new vector that has the value of this vector rotated by the rotation. Equivalent to {vec * rot}
        - Vector **ApplyRotationInvCopy**(Rotation rot); // Returns a new vector that has the value of this vector rotated by the inverse of the rotation. Equivalent to {vec / rot}
    - Operators: // Note that all of these operators do not modify any Vector values, all results are stored in new Vectors.
        - Vector * float; // Scales the current vector
        - Vector / float; // Scales the current vector
        - Vector + Vector; // Adds (sums) the two vectors together, returning the result.
        - Vector - Vector; // Subtracts the two vectors, returning the result.
        - Vector * Vector; // Vector dot product
        - Vector % Vector; // Vector cross product
        - Vector * Rotation; // Rotates the vector by the rotation, returning the resulting vector.
        - Vector / Rotation; // Rotates the vector by the inverse of the rotation, returning the resulting vector.
- ::Engine::Rotation - a unit quaternion based rotation. Note that while a unit quaternion is what is used for rotation, there is no implicit normalization - if you specify a non-unit quaternion, it will do what non-unit quaternions do: rotate **and scale** the vectors.
    - Flavors:
        - Rotation
        - Quaternion - identical to Rotation and freely convertible using implicit casts.
    - Constructors:
        - Rotation myRot(); // A unit rotation - similar to an angle of 0.
        - Rotation myRot(float yaw, float pitch, float roll); // A rotation with the given Euler parameters. The Euler angle vector (in radians) is converted to a rotation by doing the rotations around the 3 axes in Z, Y, X order.

        - Rotation myRot(Vector euler); // A rotation with the given Euler parameters. The Euler angle vector (in radians) is converted to a rotation by doing the rotations around the 3 axes in Z, Y, X order.
        - Rotation myRot(Vector axis, float angle); // A rotation with the given parameters. The angle is expressed in radians.
        - Rotation myRot(float x, float y, float z, float w); // A rotation with the given parameters.
        - Rotation myRot(Rotation); // Performs a deep copy of the passed rotation.
    - Properties: **x, y, z, w**
    - Methods:
        - Vector **ToEuler**(); // Returns the Euler representation of the rotation.
        - Vector **ToAxis**(); // Returns the axis portion of an axis-angle pair that represents the rotation.
        - float **ToAngle**(); // Returns the angle portion of an axis-angle pair that represents the rotation.
        - float **AngleTo**(Rotation other); // Returns the shortest angle between this rotation and the other.
        - void **Slerp**(Rotation other, float mix); // Changes the current rotation to a value somewhere between the current rotation and the passed rotation, based on the value of the mix parameter. If mix == 0.0f, then the resulting rotation is the same as the current rotation; if mix == 1.0f, then the resulting rotation is the same as the other rotation; if mix == 0.5f, then the resulting rotation is halfway between the two. Extreme values, eg. mix < 0.0f or mix > 1.0f, simply continue the scale past the given rotations.
        - Rotation **SlerpCopy**(Rotation other, float mix); // Returns a Rotation somewhere between the current rotation and the passed rotation, based on the value of the mix parameter. If mix == 0.0f, then the resulting rotation is the same as the current rotation; if mix == 1.0f, then the resulting rotation is the same as the other rotation; if mix == 0.5f, then the resulting rotation is halfway between the two. Extreme values, eg. mix < 0.0f or mix > 1.0f, simply continue the scale past the given rotations.
    - Operators:
        - Rotation * Rotation; // Rotates the first rotation by the second, matching the explanations given here: LSL Rotation#Combining Rotations
        - Rotation / Rotation; // Rotates the first rotation by the inverse of the second, matching the explanations given here: LSL Rotation#Combining Rotations
        - *See Vector for the application operators.*
- ::Engine::Color - a color in floating-point RGBA. Each channel's value ranges from

[0,1]. Note that this is the same as a Vector4.
    - Constructors:
        - Color myColor(); // A color with no value - black.
        - Color myColor(float r, float g, float b, float a); // A color with the given parameters.
        - Color myColor(Vector); // Performs a deep copy of the passed vector.
    - Properties: **r, g, b, a** OR **x, y, z, w**
        - Note that every value ranges from 0.0 to 1.0. For the color channels 0.0 is dark, while 1.0 is bright. For the alpha channel, 0.0 is fully transparent, while 1.0 is opaque.
- ::Engine::Entity - an abstract target for components. Note that this type is a pointer type.
    - Constructors:
        - Entity(string); // Create an entity with a specified name.
        - Entity(Entity); // Copy constructor for the pointer - does NOT copy the underlying data, it simply creates a copy of the pointer.
- ::Engine::GenericArray - a generic numerically-indexed container. Note that this type is a pointer type.
    - Constructors:
        - Envelope();
        - Envelope(Envelope); // Copy constructor for the pointer - does NOT copy the underlying data, it simply creates a copy of the pointer.
    - Methods:
        - uint **GetCount**(); // Get the count of objects stored in this Envelope.
        - bool **SaveToDisk**(string); // Attempts to write the Envelope to the file specified, overwriting anything that may have been there in the past. The Envelope is written in the NLSSD format. Updating a file can be accomplished by first loading it, making the needed changes, and then writing it to disk again. Any errors will be written to the log, and the function will return false.
        - bool **LoadFromDisk**(string); // Loads the specified file into the current Envelope. If the Envelope has data, or if the file cannot be read, the error will be written to the log, and the function will return false.
        - Append data
            - void **Add**(bool);
            - void **Add**(int);
            - void **Add**(int64);
            - void **Add**(uint);
            - void **Add**(uint64);
            - void **Add**(float);

        - void **Add**(string);
        - void **Add**(Vector);
        - void **Add**(Rotation);
        - void **Add**(Color);
        - void **Add**(Entity);
        - void **Add**(Envelope);
        - void **Add**(PropertyMap);
    - Modify data - invalid index values will cause a warning to be written to the log.
        - void **Replace**(uint index, bool);
        - void **Replace**(uint index, int);
        - void **Replace**(uint index, int64);
        - void **Replace**(uint index, uint);
        - void **Replace**(uint index, uint64);
        - void **Replace**(uint index, float);
        - void **Replace**(uint index, string);
        - void **Replace**(uint index, Vector);
        - void **Replace**(uint index, Rotation);
        - void **Replace**(uint index, Color);
        - void **Replace**(uint index, Entity);
        - void **Replace**(uint index, Envelope);
        - void **Replace**(uint index, PropertyMap);
    - Remove Data - invalid index values will cause a warning to be written to the log.
        - void **Remove**(uint index);
    - Get data - invalid index values will cause a warning to be written to the log and a default value to be returned. Attempting to get the wrong type will return a default value and cause an error to be written to the log.
        - bool **GetBool**(uint index);
        - int **GetInt**(uint index);
        - int64 **GetLong**(uint index);
        - uint **GetUInt**(uint index);
        - uint64 **GetULong**(uint index);
        - float **GetFloat**(uint index);
        - string **GetString**(uint index);
        - Vector **GetVector**(uint index);
        - Rotation **GetRotation**(uint index);
        - Color **GetColor**(uint index);
        - Entity **GetEntity**(uint index);

                - Envelope **GetEnvelope**(uint index);
                - PropertyMap **GetPropertyMap**(uint index);
- ::Engine::GenericMap - a generic string-keyed container. Note that this type is a pointer type.
    - Constructors:
        - PropertyMap();
        - PropertyMap(PropertyMap); // Copy constructor for the pointer - does NOT copy the underlying data, it simply creates a copy of the pointer.
    - Methods:
        - uint **GetCount**(); // Get the count of objects stored in this PropertyMap.
        - bool **SaveToDisk**(string); // Attempts to write the PropertyMap to the file specified, overwriting anything that may have been there in the past. The PropertyMap is written in the NLSSD format. Updating a file can be accomplished by first loading it, making the needed changes, and then writing it to disk again. Any errors will be written to the log, and the function will return false.
        - bool **LoadFromDisk**(string); // Loads the specified file into the current PropertyMap. If the PropertyMap has data, or if the file cannot be read, the error will be written to the log, and the function will return false.
        - Add/Modify data
            - void **Set**(string key, bool);
            - void **Set**(string key, int);
            - void **Set**(string key, int64);
            - void **Set**(string key, uint);
            - void **Set**(string key, uint64);
            - void **Set**(string key, float);
            - void **Set**(string key, string);
            - void **Set**(string key, Vector);
            - void **Set**(string key, Rotation);
            - void **Set**(string key, Color);
            - void **Set**(string key, Entity);
            - void **Set**(string key, Envelope);
            - void **Set**(string key, PropertyMap);
        - Remove Data - invalid key values will cause a warning to be written to the log.
            - void **Remove**(string key);
        - Get data - invalid key values will cause a warning to be written to the log and a default value to be returned. Attempting to get the wrong type will return a default value and cause an error to be written to the log.

- bool **GetBool**(string key);
- int **GetInt**(string key);
- int64 **GetLong**(string key);
- uint **GetUInt**(string key);
- uint64 **GetULong**(string key);
- float **GetFloat**(string key);
- string **GetString**(string key);
- Vector **GetVector**(string key);
- Rotation **GetRotation**(string key);
- Color **GetColor**(string key);
- Entity **GetEntity**(string key);
- Envelope **GetEnvelope**(string key);
- PropertyMap **GetPropertyMap**(string key);

Functions

- Logging
  - **::Engine::LOG**(LOG_PRIORITY, string message); // Send a message to the log file
- Callback/handle registration
  - **::Engine::RegisterEventHandler**(string handler_name, EVENT); // Registers the handler to be called when the event is fired by the engine. The handler can be any global function with the following signature: void FuncName(Envelope); Example registration: ::Engine::RegisterEventHandler("FuncName");
  - **::Engine::UnregisterEventHandler**(string handler_name, EVENT); // Unregisters the handler from being called for the given event type.
- Informational queries
  - string **::Engine::GetFolder**(SYSTEM_DIRS); // Get a known folder path.
  - array<string> **::Engine::ListActiveModules**(); // Returns an array of all the loaded module names.
  - MODULE_STATUS **::Engine::GetModuleStatus**(string name);
  - **::Engine::KillAndRestartModule**(string name);
- Engine state
  - **::Engine::Shutdown()**; // Instructs the engine to stop all activity and quit.
-