KuKu* language design goals

(* - working title)

Abstract

Easy-to-learn dev-supportive language both for
gameplay scripting with close data integration
and for store game data

I. Core

1.1. Minimalistic

Kuku should be relatively small. It must be possible to learn the main part of the language in 1 or 2 days, as it is possible for Lua. It should contain as few concepts and fundamental types as possible.

1.2. Statically typed

Script languages with really dynamic nature don't provide enough support from IDE for programers. Projects on such languages became difficult to maintain very quickly. Kuku aims to remain readable and supportive with project growth. This is why it is statically typed. It is ok to have 'Any' or 'Variant' type. But the default way to do things in kuku is specifying types.

1.3. Classes

Although classes can be simulated via simpler concepts like tables, they are so common that it is better to have it as part of language in order to have more straightforward syntax.

1.4. Value and references types

The choice of passing arguments by value or by reference in most cases depends on the type of argument. This is why kuku needs to clearly distinguish (maybe even in syntax

highlight level) between value and reference types. The meaning is the same as in C#. But we don't want any kind of passing value type instances by reference or pointers.

At the same time some scripting systems suffer from value types with a chain of accessors (a.b.c = x). Depending on whether b is a value or reference type, behaviour can vary which is very annoying. Kuku should work intuitively in both situations. Which means change the end object even in case of properties in the middle.

1.5. Garbage collector

The user should care neither about free memory nor circular reference. Which forces us to implement some kind of garbage collector. It is only affected for reference classes.

1.6 Static null checks and casting

Null-pointer dereference is one of the most common realtime errors even with statically typed languages. Kuku moves the problem to compile-time instead. This means that reference class variables can't be null. Kuku should provide nullable types instead. It also needs to contain special syntax that deal both with null checking and shadowing casted variable on successful branches. This syntax can be expanded to deal with multiple variables and class downcasting.

1.7. Parse friendly

The syntax should be easy to parse to provide Language Server Protocol functionality. We should avoid syntax that increases the count of compiler passes.

II. Integration

2.1. Bind-friendly

We want an API for seamlessly binding C++ classes to Kuku. It means that from the scripting level they should look like normal kuku classes in all contexts.

If C++ class is bound to a *value kuku class* then its instances can be passed by value and should be placed at the virtual machine stack. Passing by pointer or reference must lead to a new instance copy anyway.

If C++ class is bound to a *reference kuku class* then it can be passed by shared_ptr. This pointer will be released when the corresponding kuku variable is garbage collected. If an instance of such class is passed by value then it should be moved to the heap and be controlled via shared_ptr anyway.

Passing by raw pointer or reference for such a class might be supported by kuku but not recommended because raw pointers can lead to memory corruption. Users should avoid

such practice except they know for sure that the lifetime of passed objects is longer enough (e.g. global variables). This is why kuku should generate at least a warning.

2.2. Data format oriented

Kuku is being designed to be able to express domain specific hierarchy-organized data directly by language subset. Syntax for it should be the same with object initializing of corresponding types in memory, including C++ class bindings.

III. Serialization

3.1. Built-in serialization mechanism

Class definition syntax should contain explicit way to mark fields and properties to be or not to be serialized. Built-in serializer should use this information when stringifying an object. The resulting output should be written in a form of valid kuku code (its data subset). Execution of this code should produce a restored object.

3.2. Circle reference resolving

Reference-class instances should not be written twice during serialization. Kuku should support special syntax to refer to objects that were met earlier. Since serialization format is hierarchy-organized, this syntax can be similar to relative path in order to minimize merge conflicts.

3.3. Binary format

Human-readable merge-friendly text format is useful during development but it can cause performance issues in a final product. Thus the serializer should not work directly with a text. Instead, we would like to use a binary format that is able to be unambiguously converted to text and vice versa in the separate step. It is also important that the binary format should not contain names of fields more than one time.

3.4. Versioning support

Serialization systems are often very restrictive for renaming. It usually leads to inconsistent naming and complicates refactoring. Kuku uses another approach. It requires every field and class to be annotated with an unique randomly generated guid. Guids could be generated by intellisense but they are stored in a separate file (huge table which map fullnames to guids).

Binary format should contain field guids in addition to field names to stay compatible. On the other hand text format can omit guids because text format is used only during the development and always related to specific codebase state.

IV. Parallelization

4.1. Coroutines

Coroutines are really game changer in asynchronous programming. It is a must have feature for gameplay scripting. Although we don't want to introduce invasive keywords to mark async functions and calls. Any kuku function should support yielding from any place. Yield should lead to the error if calling from not-coroutine context. But the existence of yield by itself shouldn't prevent the call of function.

It would be nice to have a mechanism of saving coroutine stack to be restorable at least for the current application version. But it is expected to be very limited for versioning.

4.2. Object level locks

Some scriptable languages provide support for multithreading but actually degrade to one thread because they use virtual-machine level locks. Kuku should use object-level locks if any.

V. Infrastructure

5.1. Byte code

Kuku needs a compiler that converts sources to intermediate bytecode. This byte code is very likely to be the same as the binary format of the serializer. Basically it means that converting data from text to binary is correspond to just script compiling. And decompiling for the back process.

5.3. Data autocomplete

Since kuku can treat data as part of the codebase, it is expected to have some advanced autocomplete about static data variables.

5.2. Reflection

We need the ability to introspect any kuku object. It is especially useful for IDE editors.

5.4. Debugger

Kuku needs to provide basic C++ API for placing breakpoints at specific lines and for stack examination.

5.5. Strict codestyle

Kuku is likely to force some codestyle at syntax level. For instance, it can require specific naming rules for local variables, class fields or function arguments.

5.6. C++ bind code generator

A separate tool to generate binding code for C++ is very desirable. Most likely it will be based on parsing c++11 attribution.