Blueprint Compiler Overview

The steps of the Blueprint compilation process

Blueprints, like regular C++ classes, need to be compiled before they can be used in-game. When you hit the **Compile** button in the **Blueprint Editor**, the process of converting the properties and graphs of a Blueprint asset into a class is performed.

Terminology

$ FKismetCompilerContext : The class that does the work of compilation. A new instance is spawned for each compile. Stores reference to the class being compiled, the blueprint, etc.

$ FKismetFunctionContext : Holds the information for compiling a single function, like a reference to the associated graph, properties, and generated UFunction.

$ FNodeHandlingFunctor : A helper class that handles processing one class of node in the compiler (a singleton). Contains functions for registering pin connections, and generating compiled statements.

$ FKismetCompiledStatement : Unit of work in the compiler. The compiler translates nodes into a set of compiled statements, which the backend translates into bytecode operations.

Examples: Variable assignment, Goto, Call

$ FKismetTerm : A terminal in the graph (literal, const, or variable reference). Each data pin connection is associated with one of these! You can also make your own terms in a `NodeHandlingFunctor` for scratch variables, intermediate results, etc.

Compilation Process

The basic process of compiling a Blueprint is outlined below:

(Click item to view)

1. Clean the Class
2. Create Class Properties
3. Create Function List
4. Bind and Link the Class
5. Compile Functions*
6. Finish Compiling Class
7. Backend Emits Generated Code*
8. Copy Class Default Object Properties
9. Reinstance

Clean the Class

Classes are compiled in place, which means the same **UBlueprintGeneratedClass** is cleaned and reused over and over, so that pointers to the class do not have to be fixed up. **CleanAndSanitizeClass()** moves properties and functions off the class and into a trash class in the transient package, and then clears any data on the class.

Create Class Properties

The compiler iterates over the Blueprint's **NewVariables** array, as well as some other places (construction scripts, etc.) to find all of the UProperties needed by the class and then creates UProperties on the UClass's scope in the function **CreateClassVariablesFromBlueprint()**.

Create Function List

The compiler creates the function list for the class by processing the event graphs, processing the regular function graphs, and *pre-compiling* the functions, i.e. calling **PrecompileFunction()** for each context.

Process Event Graphs

Processing of the event graphs is performed by the **CreateAndProcessUberGraph()** function. This copies all event graphs into one big graph, after which nodes are given a chance to expand. Then, a function stub is created for each Event node in the graph, and an **FKismetFunctionContext** is created for each event graph.

Process Function Graphs

Processing of the regular function graphs is done by the **ProcessOneFunctionGraph()** function, which duplicates each graph to a temporary graph where nodes are given a chance to expand. A **FKismetFunctionContext** is created for each function graph as well.

Pre-compile Functions

Pre-compiling of the functions is handled by the **PrecompileFunction()** of each context. This function performs the following actions:

- Schedules execution and calculates data dependencies.
- Prunes any nodes that are unscheduled or not a data dependency.
- Runs the node handler's **RegisterNets()** on each remaining node.
- This creates the **FKismetTerms** for values within the function.
- Creates the **UFunction** and associated properties.

Bind and Link the Class

Now that the compiler is aware of all of the UProperties and UFunctions for the class, it can bind and link the class, which involves filling out the property chain, the property size,

function map, etc. At this point, it essentially has a class header - minus the final flags and metadata - as well as a Class Default Object (CDO).

Compile Functions

The next step consists of generating **FKismetCompiledStatment** objects for the remaining nodes which is accomplished through the node handler's **Compile()** function, using **AppendStatementForNode()**. This function can create **FKismetTerm** objects in the compile function as long as they are only used locally.

Finish Compiling Class

To finish compiling the class, compiler finalizes the class flags and propagates flags and metadata from the parent class before finally performing a few final checks to make sure everything went alright in the compile.

Backend Emits Generated Code

The backends convert the collection of statements from each function context into code. There are two backends in use:

- **FKismetCompilerVMBackend** - Converts FKCS to UnrealScript VM bytecode which are then serialized into the function's script array.
- **FKismetCppBackend** - Emits *C++-like* code for debugging purposes only.

Copy Class Default Object Properties

Using a special function, **CopyPropertiesForUnrelatedObjects()**, the compiler copies the values from the old CDO of the class into the new CDO. Properties are copied via tagged serialization, so as long as the names are consistent, they should properly be transferred. Components of the CDO are re-instanced and fixed up appropriately at this stage. The GeneratedClass CDO is authoritative.

Re-instance

Since the class may have changed size and properties may have been added or removed, the compiler needs to re-instance all objects with the class that were just compiled. This process uses a **TObjectIterator** to find all instances of the class, spawn a new one, and then uses the **CopyPropertiesForUnrelatedObjects()** function to copy from the old instance to the new one.

For details, see the **FBlueprintCompileReinstancer** class.