Documentation: KCurveCommander

By Robert Gervais for Demon Crush

Overview

KCurveCommander (or simply KMath's "Curve Commander") is a curve command interface feature that was added to KMath. As a reminder, KMath is located under the following Relative Path.

```
..\..\Source\ProjectK\KMath.h
...\..\Source\ProjectK\KMath.cpp
```

And KMath is largely responsible for performing computations on constant expressions, which is explained at a high level in this blog post. KMath also computes the result of Boolean expressions and also inputs what we call "non-constant" expressions, which are semantically natural language expressions in the form of variable inputs for computational output.

One additional feature request was to add a way to look up the y-value of an x-input on a predefined curve, like in the following cartoon drawing.

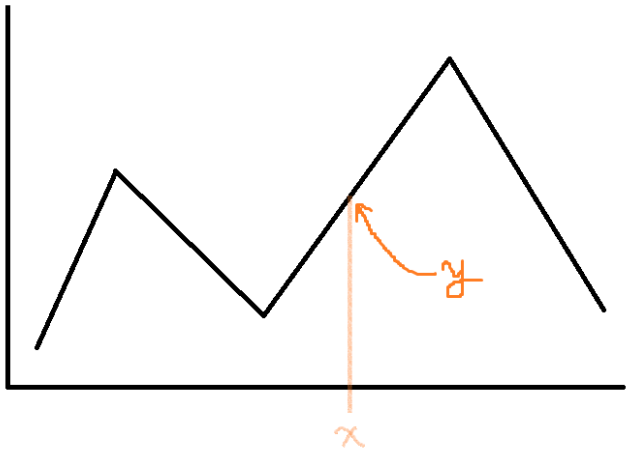


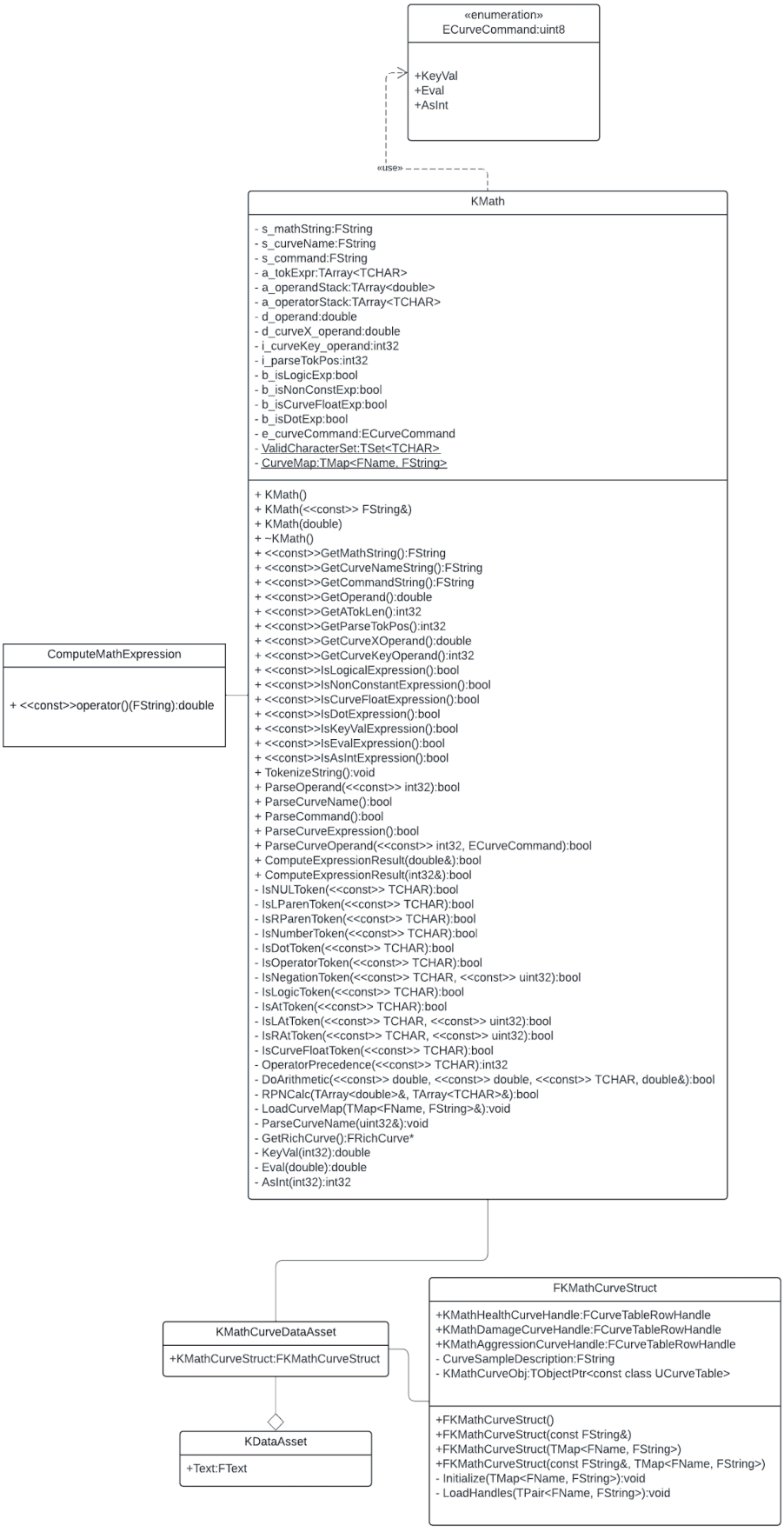
Figure 1: This curve could represent any gameplay variable, such as Health or Damage.

Continue reading the next section to learn how the feature was implemented.

Technical Implementation

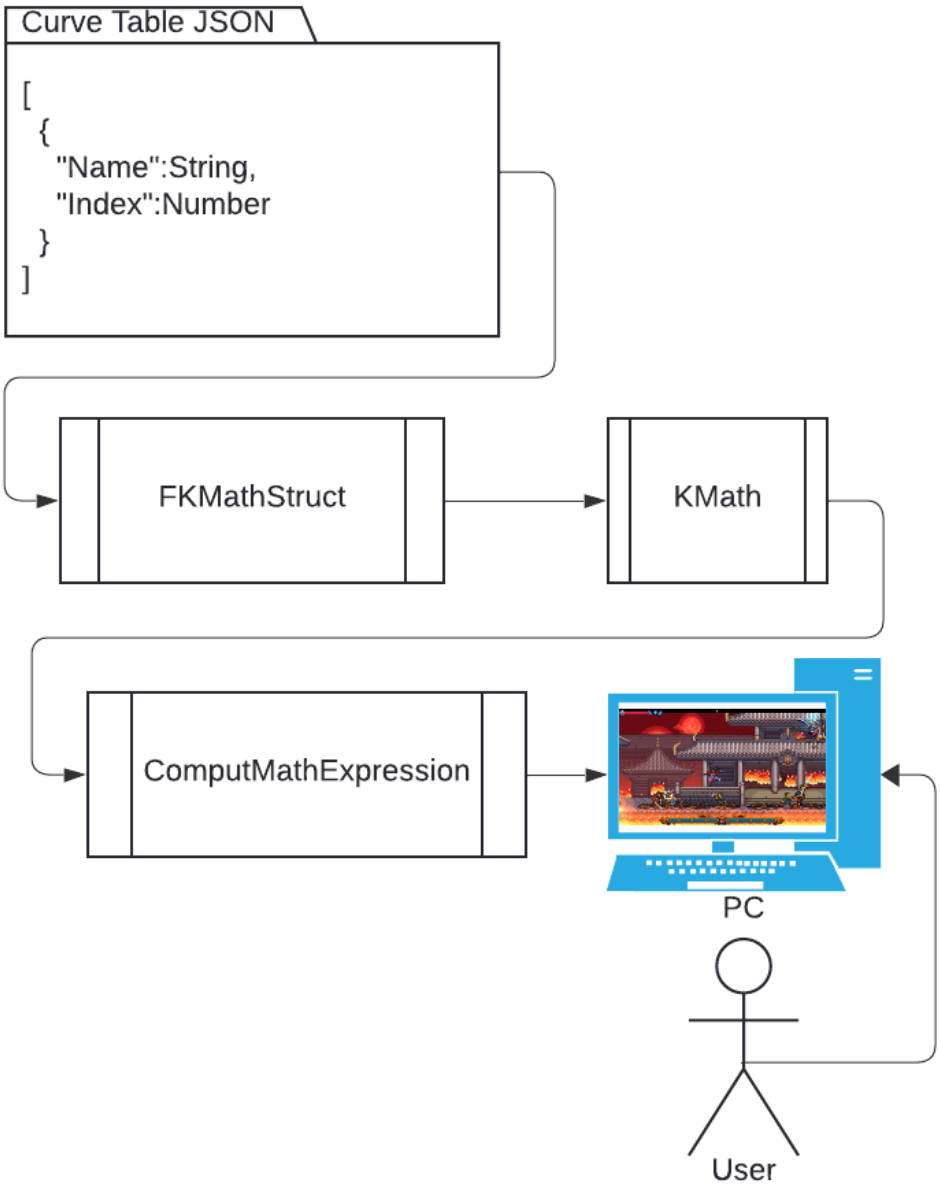
UML Diagram

Here is the UML diagram and a link to the UML in Lucidchart.


«enumeration»
ECurveCommand:uint8
+KeyVal
+Eval
+AsInt
«use»
KMath
- s_mathString:FString
- s_curveName:FString
- s_command:FString
- a_tokExpr:TArray<TCHAR>
- a_operandStack:TArray<double>
- a_operatorStack:TArray<TCHAR>
- d_operand:double
- d_curveX_operand:double
- i_curveKey_operand:int32
- i_parseTokPos:int32
- b_isLogicExp:bool
- b_isNonConstExp:bool
- b_isCurveFloatExp:bool
- b_isDotExp:bool
- e_curveCommand:ECurveCommand
- ValidCharacterSet:TSet<TCHAR>
- CurveMap:TMap<FName, FString>
+ KMath()
+ KMath(<<const>> FString&)
+ KMath(double)
+ ~KMath()
+ <<const>>GetMathString():FString
+ <<const>>GetCurveNameString():FString
+ <<const>>GetCommandString():FString
+ <<const>>GetOperand():double
+ <<const>>GetATokLen():int32
+ <<const>>GetParseTokPos():int32
+ <<const>>GetCurveXOperand():double
+ <<const>>GetCurveKeyOperand():int32
+ <<const>>IsLogicalExpression():bool
+ <<const>>IsNonConstantExpression():bool
+ <<const>>IsCurveFloatExpression():bool
+ <<const>>IsDotExpression():bool
+ <<const>>IsKeyValExpression():bool
+ <<const>>IsEvalExpression():bool
+ <<const>>IsAsIntExpression():bool
+ TokenizeString():void
+ ParseOperand(<<const>> int32):bool
+ ParseCurveName():bool
+ ParseCommand():bool
+ ParseCurveExpression():bool
+ ParseCurveOperand(<<const>> int32, ECurveCommand):bool
+ ComputeExpressionResult(double&):bool
+ ComputeExpressionResult(int32&):bool
- IsNULToken(<<const>> TCHAR):bool
- IsLParenToken(<<const>> TCHAR):bool
- IsRParenToken(<<const>> TCHAR):bool
- IsNumberToken(<<const>> TCHAR):bool
- IsDotToken(<<const>> TCHAR):bool
- IsOperatorToken(<<const>> TCHAR):bool
- IsNegationToken(<<const>> TCHAR, <<const>> uint32):bool
- IsLogicToken(<<const>> TCHAR):bool
- IsAtToken(<<const>> TCHAR):bool
- IsLAtToken(<<const>> TCHAR, <<const>> uint32):bool
- IsRAtToken(<<const>> TCHAR, <<const>> uint32):bool
- IsCurveFloatToken(<<const>> TCHAR):bool
- OperatorPrecedence(<<const>> TCHAR):int32
- DoArithmetic(<<const>> double, <<const>> double, <<const>> TCHAR, double&):bool
- RPNCalc(TArray<double>&, TArray<TCHAR>&):bool
- LoadCurveMap(TMap<FName, FString>&):void
- ParseCurveName(uint32&):void
- GetRichCurve():FRichCurve*
- KeyVal(int32):double
- Eval(double):double
- AsInt(int32):int32
ComputeMathExpression
+ <<const>>operator()(FString):double
KMathCurveDataAsset
+KMathCurveStruct:FKMathCurveStruct
KDataAsset
+Text:FText
FKMathCurveStruct
+KMathHealthCurveHandle:FCurveTableRowHandle
+KMathDamageCurveHandle:FCurveTableRowHandle
+KMathAggressionCurveHandle:FCurveTableRowHandle
- CurveSampleDescription:FString
- KMathCurveObj:TObjectPtr<const class UCurveTable>
+FKMathCurveStruct()
+FKMathCurveStruct(const FString&)
+FKMathCurveStruct(TMap<FName, FString>)
+FKMathCurveStruct(const FString&, TMap<FName, FString>)
- Initialize(TMap<FName, FString>):void
- LoadHandles(TPair<FName, FString>):void

At a high level, KCurveCommander is implemented as an extension of `KMath` with `FKMathCurveStruct` to support loading curve data to assemble an FRichCurve that can be read at real-time. Additionally, `ECurveCommand` helps enumerate routine execution so that `KMath` can output a result with `ComputeMathExpression`.

And to capture the high-level operation of KCurveCommander it's useful to think of KCurveCommander as the assembly step of the input-output pipeline, which is illustrated by this helpful diagram.



User Input

From a user's perspective, `KCurveCommander` is implemented such that a User can pass one of two input value types (generally represented as x-value on a 2D graph) to `KMath` in order to return the queried y-value. The User can pass curve expressions to `KMath` to query a pre-defined curve. For example, let's say a User wants to query their Health curve using any of the allowable syntactical patterns, which are represented in this table.

Curve expression	Expression description
`"@Health@#1"`	User is looking up Health y-value at key (or index) 1. Note that this is not a dot expression, which means that `KMath` will use the default behavior of passing the key as an `int32` to return a y-value as a `double` from the curve.
`"@Health.KeyVal@#2"`	User is looking up Health y-value at key (or index) 2. Note that this is a dot expression, which means that `KMath` will explicitly pass the key as an `int32` to return a y-value as a `double` from the curve.
`"@Health.Eval@#3.4"`	User is looking up Health y-value at a time-value of 3.4. Note that this is a dot expression, which means that `KMath` will explicitly pass the time-value as a `double` to return a y-value as a `double` from the curve.
`"@Health.AsInt@#5"`	User is looking up Health y-value at key (or index) 5. Note that this is not a dot expression, which means that `KMath` will use the default behavior of passing the key as an `int32` to return a y-value as an `int32` from the curve.

This query syntax is supported by the `ECurveCommand` enum class at the top of `KMath.h`, with the enumerations described as follows.

enum	description
`KeyVal`	Pass the key (`int32`) to return a value (`double`) from the rich curve.
`Eval`	Pass an x-value (`double`) to evaluate the rich curve and return a y-value (`double`) at the given time.
`AsInt`	Pass an key (`int32`) to return a value (`int32`) from the rich curve.

Usage of `KCurveCommander` within `KMath` is governed by these three enumerated behaviors, and there is a private variable in `KMath` with the name `e_curveCommand`, which is used in `KMath` for the `ParseCommand` and `ComputeExpression` routines. The `ECurveCommand` enum is used in `KMath` for the `ParseCurveOperand` routine.

We'll cover basic and advanced usage of `KCurveCommander` but before we do, the extension works on a basic `Load-Read-Write` pattern, which is described as follows.

Step	Function
1 (Load)	`KMath::LoadCurveMap`
2 (Read)	`KMath::GetRichCurve`

3 (Write)	`KMath::KeyVal`
	`KMath::Eval`
	`KMath::AsInt`

Load

`KMath::LoadCurveMap` loads JSON curve tables from a directory and assembles them into a `TMap`. `KMath::LoadCurveMap` is called by `KMath::GetRichCurve`.

Read

`KMath::LoadCurveMap` gets called by `KMath::GetRichCurve`, which 1) Loads a CurveMap, 2) assembles a `CurveStruct` by passing the loaded curve map to the `FKMathCurveStruct` constructor and if found, 3) returns a `CurveHandle`. `KMath::GetRichCurve` calls `KMath::LoadCurveMap` to assemble a `CurveStruct` in order to return a `CurveHandle`. This function is called by `KeyVal`, `Eval`, and `AsInt`.

Write

`KeyVal`, `Eval`, and `AsInt` are routines that write output values depending on the queried index used to read into a `CurveHandle`, which is read by `KMath::GetRichCurve`.

KeyVal

`KMath::KeyVal` calls `KMath::GetRichCurve` to get an `FKeyHandle`, which is an index that enables this function to query the `FRichCurve` and return the query value as a double return type.

Eval

`KMath::Eval` calls `KMath::GetRichCurve` to `BakeCurve` at a sample rate of `1.0` before returning the query value as a `double` return type.

AsInt

`KMath::KeyVal` calls `KMath::GetRichCurve` to get an `FKeyHandle`, which is an index that enables this function to query the `FRichCurve` and `StaticCast` the query value to an `int32` return type.

Parsing User Input

A dependency to support KCurveCommander's Load-Read-Write pattern is KMath's parse feature. In order to parse user input, KMath follows a lightweight command syntax with the following Extended Backus-Naur Form (EBNF) grammar.

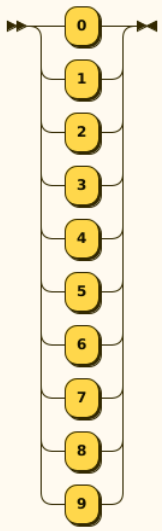
Lightweight Command Syntax EBNF

```
Digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Integer ::=  Digit |  Digit+
FloatingPoint ::= (integer)'.'(integer)
Command ::= "KeyVal" | "Eval" | "AsInt"
CurveName ::= "Health" | "Damage" | "Aggression"
CurveExpression ::= '@'(CurveName)'.'(Command)'@' | '@'(CurveName)'@'
CommandExpression ::= ('"'CurveExpression'#'Integer'"') | ('"'CurveExpression'#'FloatingPoint'"')
```

Lightweight Command Syntax Railroad Diagram

Digit Railroad

Digit:



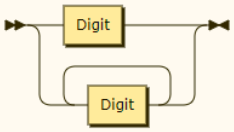

```
Digit    ::= '0'
           | '1'
           | '2'
           | '3'
           | '4'
           | '5'
           | '6'
           | '7'
           | '8'
           | '9'
```

referenced by:

- Integer

Integer and FloatingPoint Railroads

Integer:




```
Integer  ::= Digit
           | Digit+
```

referenced by:

- CommandExpression

FloatingPoint:

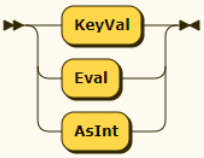

```
FloatingPoint
         ::= integer '.' integer
```

referenced by:

- CommandExpression

Command and CurveName Railroads

Command:




```
Command  ::= 'KeyVal'
           | 'Eval'
           | 'AsInt'
```

referenced by:

- CurveExpression

CurveName:

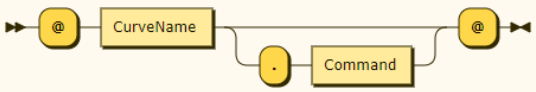

```
CurveName
          ::= 'Health'
            | 'Damage'
            | 'Aggression'
```

referenced by:

- CurveExpression

CurveExpression and CommandExpression Railroads

CurveExpression:

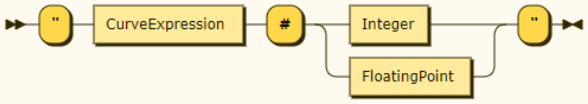



```
CurveExpression
        ::= '@' CurveName ( '.' Command )? '@'
```

referenced by:

- CommandExpression

CommandExpression:




```
CommandExpression
        ::= '"' CurveExpression '#' ( Integer | FloatingPoint ) '"'
```

no references

As you can see from the `CommandExpression`, the `'#'` token indicates that this is a valid command expression which `KMath` can parse a given `CurveName` or a `CurveExpression`. And during the tokenization step, `KMath::IsCurveFloatToken` tests whether a given `CommandExpression` contains the `'#'` token. From an architectural level, KMath performs the following routines when parsing a `CommandExpression`.

Step	Function(s)	Description
Tokenize	`KMath::TokenizeString`	During the tokenization step, `KMath` detects the `'#'` token, which indicates that this is a `CurveFloatToken` and is therefore a `CurveExpression` that needs to be parsed.

Parse	`KMath::ParseCurveName`
	`KMath::ParseCurveExpression`
	`KMath::ParseCommand`
	`KMath::ParseCommandOperand`
		`ParseCurveName` then parses out the `CurveName` from the `CurveExpression` using the EBNF grammar rule before `ParseCurveExpression` runs `ParseCommand` and `ParseCommandOperand` to determine which `Command` should operate on the given `CommandOperand`, which can either be an `int32` or a `float`(ing point) value as defined by the EBNF grammar.

Evaluate	`KMath::KeyVal`
	`KMath::Eval`
	`KMath::AsInt`
		Depending on what `KMath::ParseCommand` detects, one of these three functions will read the y-value

		from one of the stored curves returned as `FRichCurve*` from `KMath::GetRichCurve`.

Evaluating User Input

So to bring it all together, we've extended ComputeMathExpression::operator()(FString) to encapsulate the Tokenize-Parse-Evaluate pattern with the following highlighted (new) code.

```
class PROJECTK_API ComputeMathExpression
{
public:
	double operator()(FString mS) const
	{
		KMath kMathObj = KMath(mS);
		//TOKENIZE
		kMathObj.TokenizeString();

		double d_result = 0.0;
		//IF '#' TOKEN DETECTED DURING TOKENIZATION
		if (kMathObj.IsCurveFloatExpression())
		{
			//PARSE CURVE EXPRESSION TO EXTRACT CURVE NAME, COMMAND, AND OPERAND
			kMathObj.ParseCurveExpression();
			//EVALUATE AND RETURN Y-VALUE FROM CURVE
			if (kMathObj.IsKeyValExpression() || kMathObj.IsEvalExpression())
			{
				const bool bSuccess = kMathObj.ComputeExpressionResult(d_result);
				check(bSuccess);
				return d_result;
			}
			if (kMathObj.IsAsIntExpression())
			{
				int32 i_result;
				const bool bSuccess = kMathObj.ComputeExpressionResult(i_result);
				check(bSuccess);
				return i_result;
			}
		}
		const bool bSuccess = kMathObj.ComputeExpressionResult(d_result);
		check(bSuccess);
		return d_result;
	}
};
```

To get started, we recommend following the basic and advanced usage tutorials before learning how to set up new curves. Keep reading to learn more.

Basic Usage

KMathTests.hpp contains unit tests that demonstrate the basic usage of KCurveCommander under KMathTests::ParseStringUnit::Test0, which is the base case for parsing a CurveName and the index to look-up in the curve being referenced.

For testing purposes, three sample curves have been provided in order to test the base use case of looking up a y-value in the CurveName being indexed. The three sample curves are Health, Damage, and Aggression. To test the basic usage, pass a string that follows this syntactic rule.

```
"CurveName#Index"
```

Health Curve Example

An example of this basic usage syntax is passing "@Health@#1". Under the hood, KMath will run KMath::TokenizeString() and detect the '#' token with KMath::IsCurveFloatToken() predicate, and then it will run KMath::ParseCurveExpression(), which will not detect the '.' token with KMath::IsDotExpression() predicate, and then it will call KMath::ParseCurveName() in order to detect the Health curve name before running KMath::ParseCurveOperand() to detect the '1' int32 index value, which it uses to look-up the y-value in the Health curve using KMath::KeyVal(), which is the default command associated with this basic usage.

Advanced Usage

KMathTests.hpp contains unit tests that demonstrate advanced usage of KCurveCommander under KMathTests::ParseStringUnit::Test1, which is the advanced case for parsing a CurveName and Command along with either and index or a time value to look-up in the curve being referenced.

For testing purposes, three sample curves have been provided in order to test the advanced use case of looking up a y-value in the CurveName being indexed. The three sample curves are Health, Damage, and Aggression. To test the advanced usage, pass a string that follows this syntactic rule.

```
"CurveName.Command#Index"
```

Damage Curve Example

An example of this advanced usage syntax is passing "`@Damage.Eval@#3.4`". Under the hood, `KMath` will run `KMath::TokenizeString()` and detect the `'#'` token with `KMath::IsCurveFloatToken()` predicate, and then it will run `KMath::ParseCurveExpression()`, which will detect the `'.'` token with `KMath::IsDotExpression()` predicate, and then it will call `KMath::ParseCurveName()` in order to detect the `Damage` curve name before running `KMath::ParseCurveOperand()` to detect the `'3.4' double` time value, which it will use to look-up the y-value in the `Damage` curve using the `KMath::Eval()` function, which will call the `BakeCurve()` function associated with this advanced usage.

Aggression Curve Example

Another example of this advanced usage syntax is passing "`@Aggression.AsInt@#1`". Under the hood, `KMath` will run `KMath::TokenizeString()` and detect the `'#'` token with `KMath::IsCurveFloatToken()` predicate, and then it will run `KMath::ParseCurveExpression()`, which will detect the `'.'` token with `KMath::IsDotExpression()` predicate, and then it will call `KMath::ParseCurveName()` in order to detect the `Aggression` curve name before running `KMath::ParseCurveOperand()` to detect the `'1' int32` index value, which it will use to look-up the y-value in the `Aggression` curve using the `KMath::AsInt()` function, which will `StaticCast` the return type to `int32` data type.

Note that there are several other examples that can be stepped through under `AProjectKGameMode::TestKMath()`, tests k thru q.

After checking out all of the examples, keep reading to learn how to set up new curves.

How to Set up New Curves

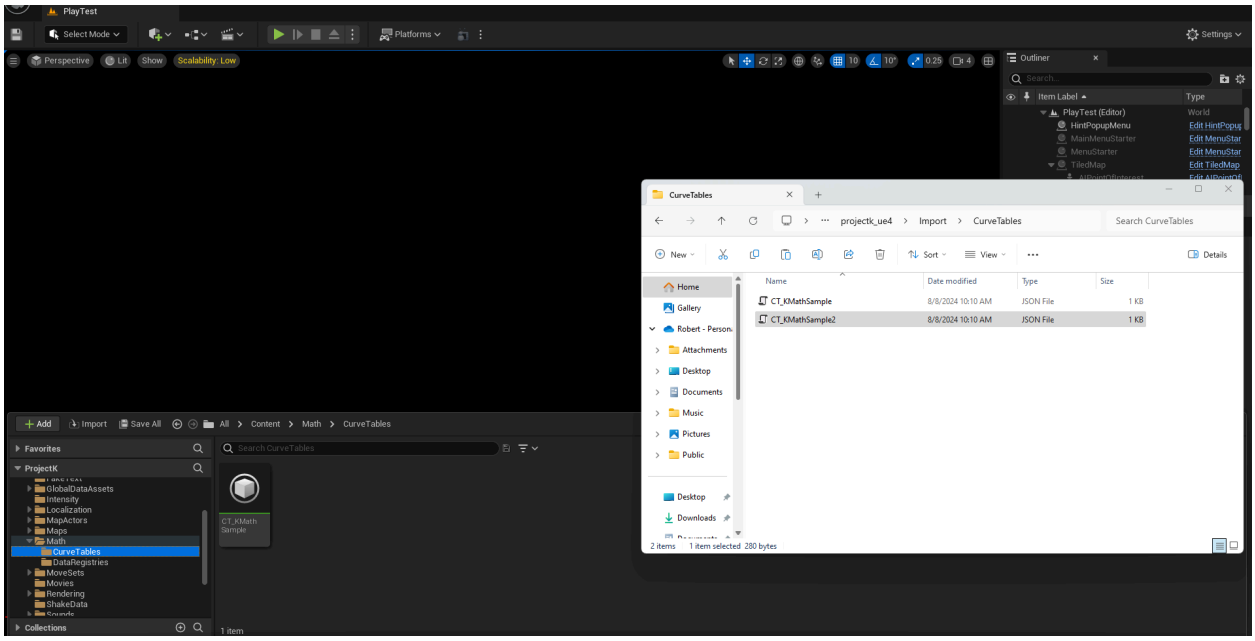
Currently, there are three sample curves for testing purposes – namely `Health`, `Damage`, and `Aggression` curves, which can be found under `Import/CurveTables/CT_KMathSample.JSON`, which can be opened to learn about the JSON formatting rules. Follow these steps to set up new curves.

Steps

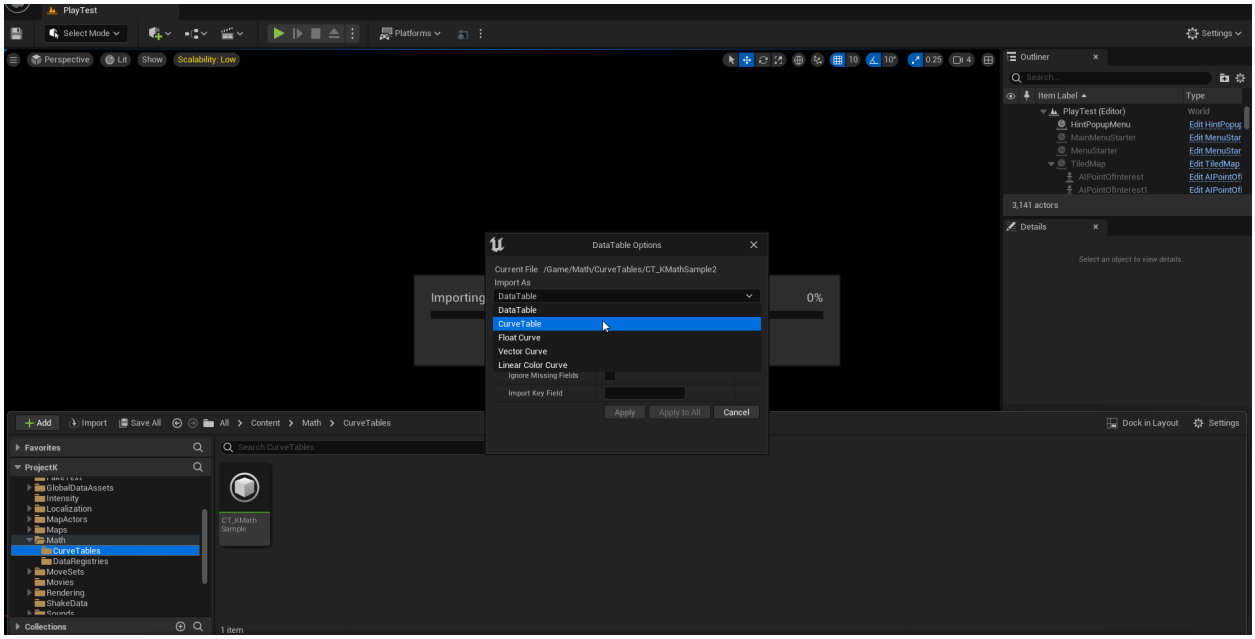
1. Open `CT_KMathSample.JSON` to learn how to format the JSON file.

2. Create a new JSON file named `CT_KMath<Name>.JSON` and save it under the `Import/CurveTables` directory.

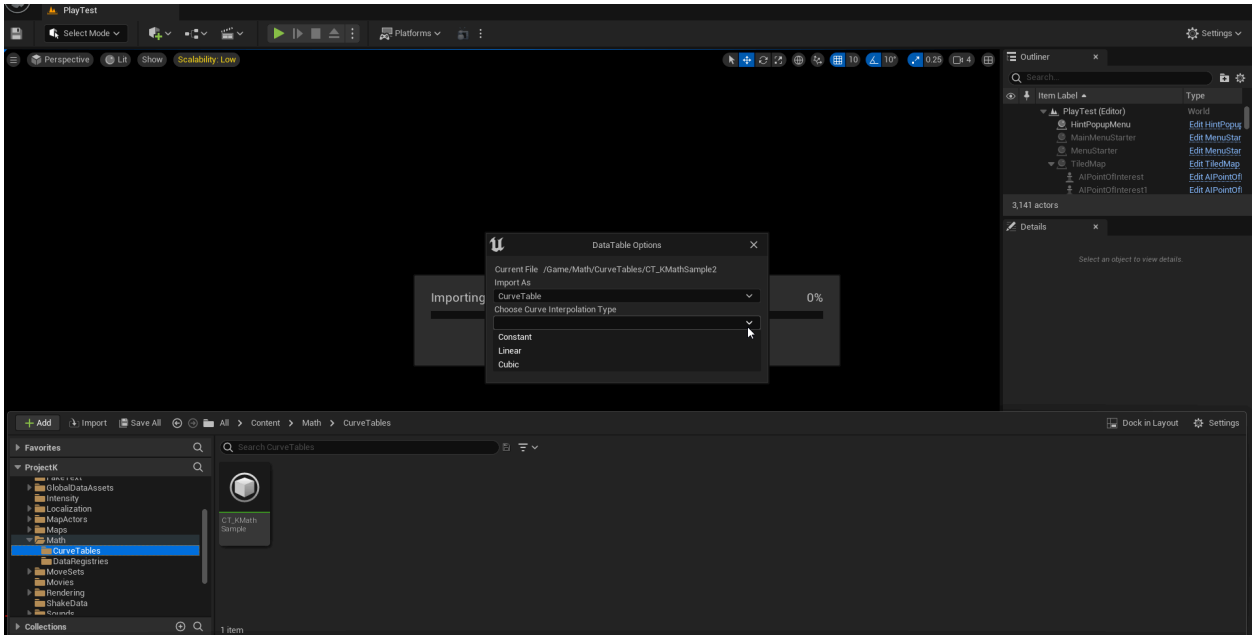
3. Drag the JSON file into the `Math/CurveTables` Content directory.



4. Import As `CurveTable`.



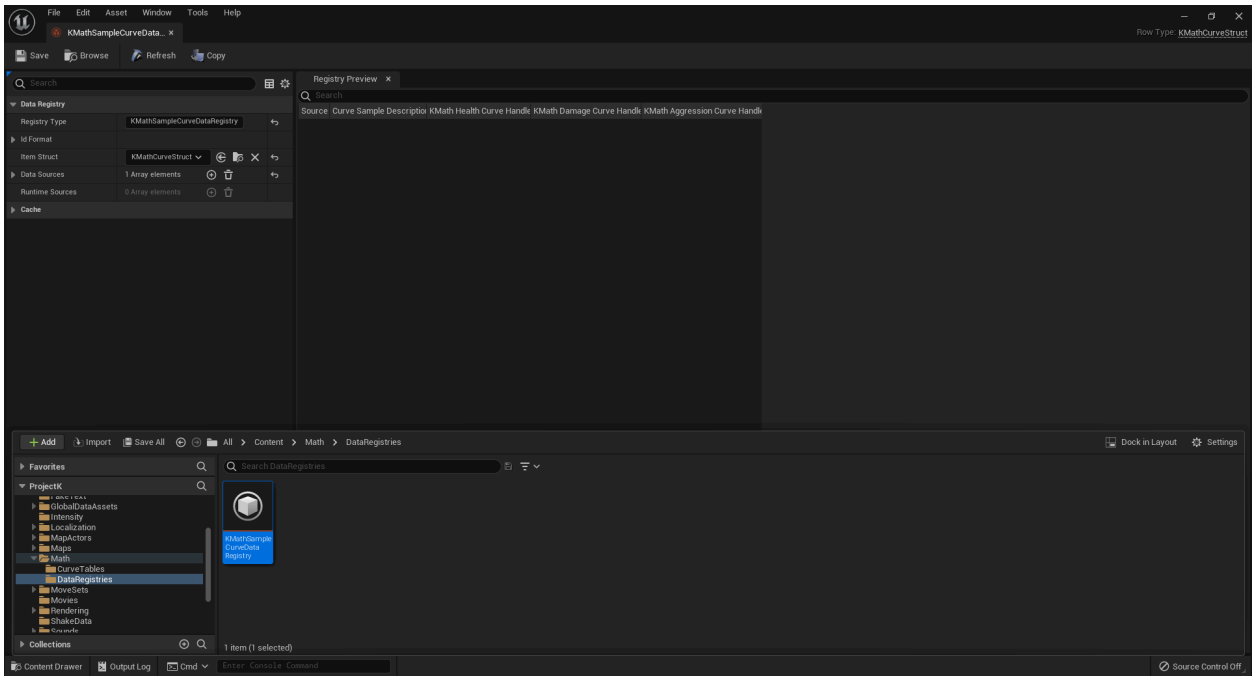
5. Choose the Curve Interpolation Type.



`Constant` values in the y-coordinate won't interpolate between values in the x-coordinate, and will clamp to the previous known value of x. `Linear` values in the y-coordinate will linearly interpolate between values in the x-coordinate (think of a linear line graph). `Cubic` values in the y-coordinate will smoothly interpolate between values in the x-coordinate (think of a cubic graph).

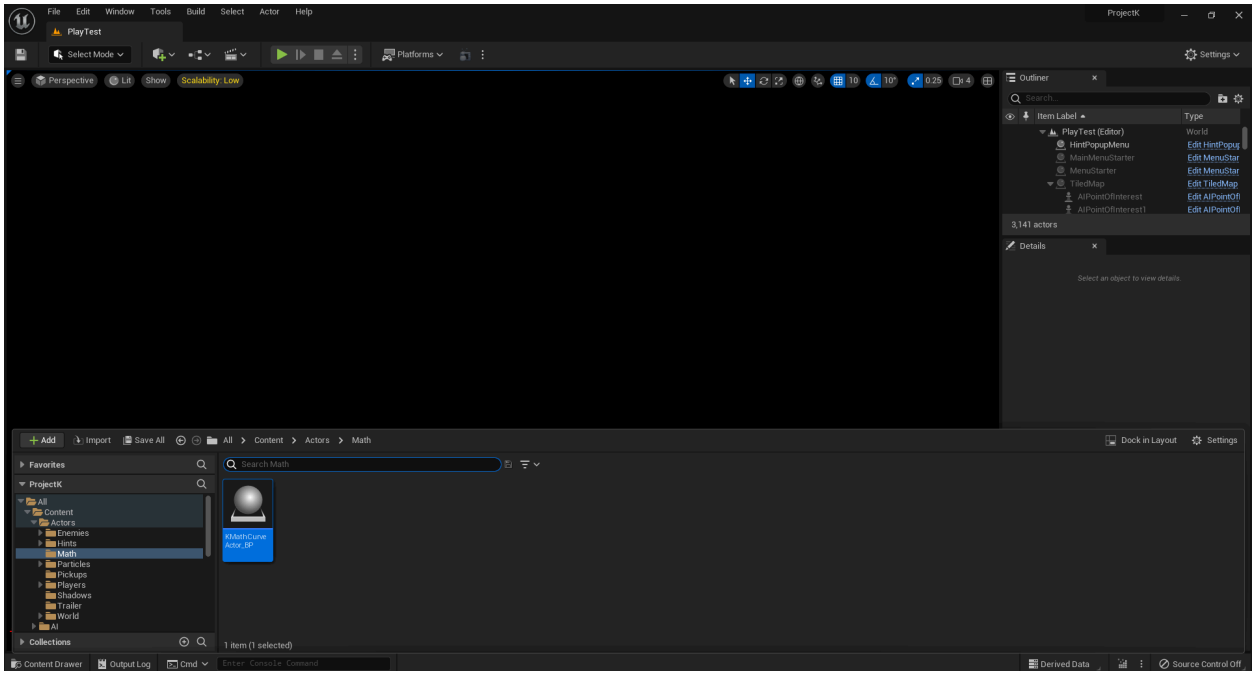
6. Declare `FCurveTableRowHandle` in `KMathCurveDataAsset.h`
7. Set the `pair.Key` value pairs to the appropriate key name under `FKMathCurveStruct::LoadHandles()`
8. Set the appropriate `curveTablePath` and `curveName` and assemble the `curveMap` under `KMath::LoadCurveMap()`
9. Update `KMath::GetRichCurve()` so that it returns a pointer to the appropriate `FRichCurve`.
10. Set up a new `KMath<name>CurveRegistry` under, making sure to follow the example set up in `DataRegistry'/Game/Math/DataRegistries/KMathSampleCurveDataRegistry.KMa`

`thSampleCurveDataRegistry'`

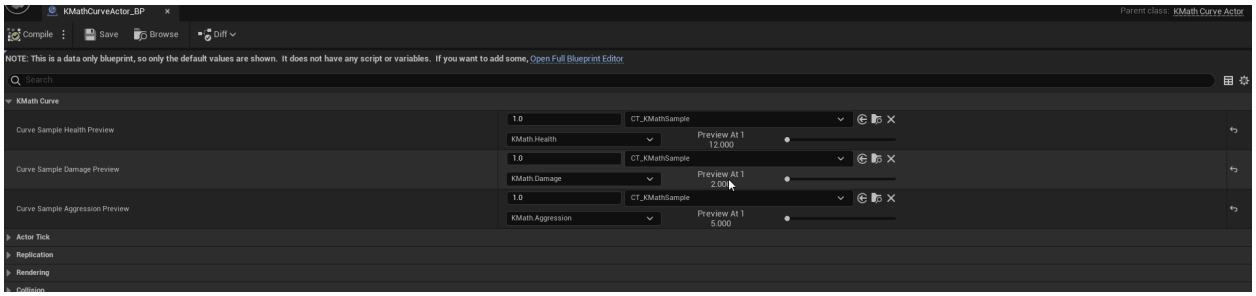


Previewing New Curves

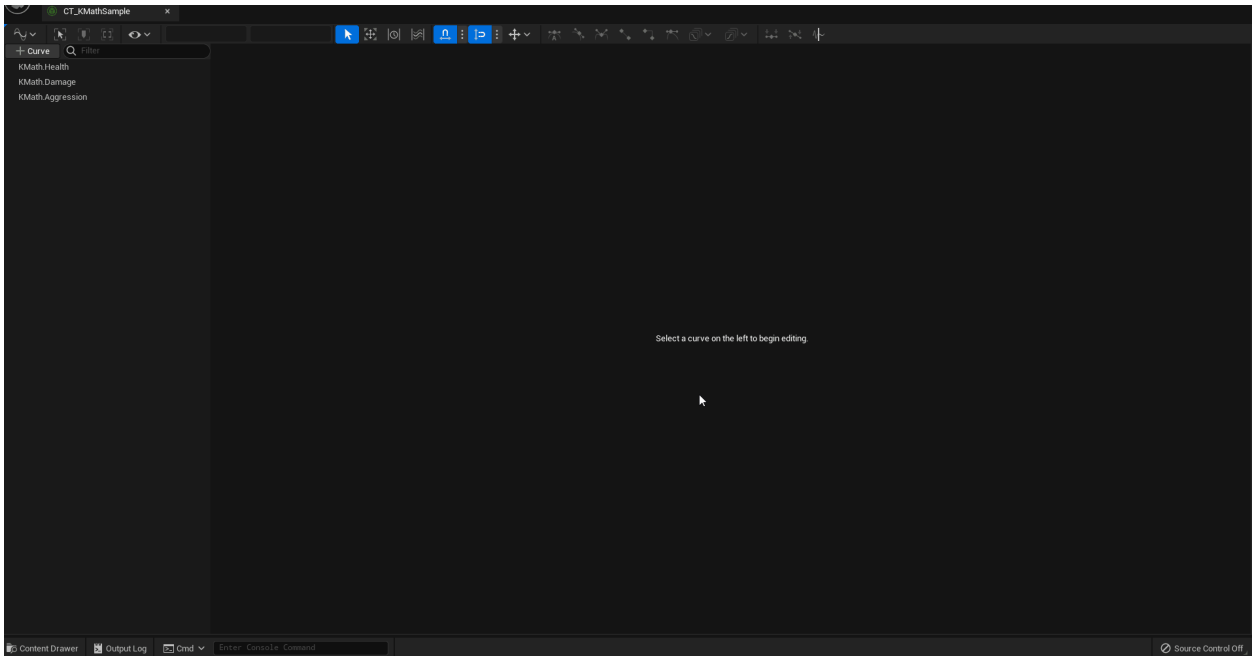
If you want to preview your new curves, you can create a new Blueprint Actor under the `Actors/Math` Content directory like the example found under `Blueprint'/Game/Actors/Math/KMathCurveActor_BP.KMathCurveActor_BP'`



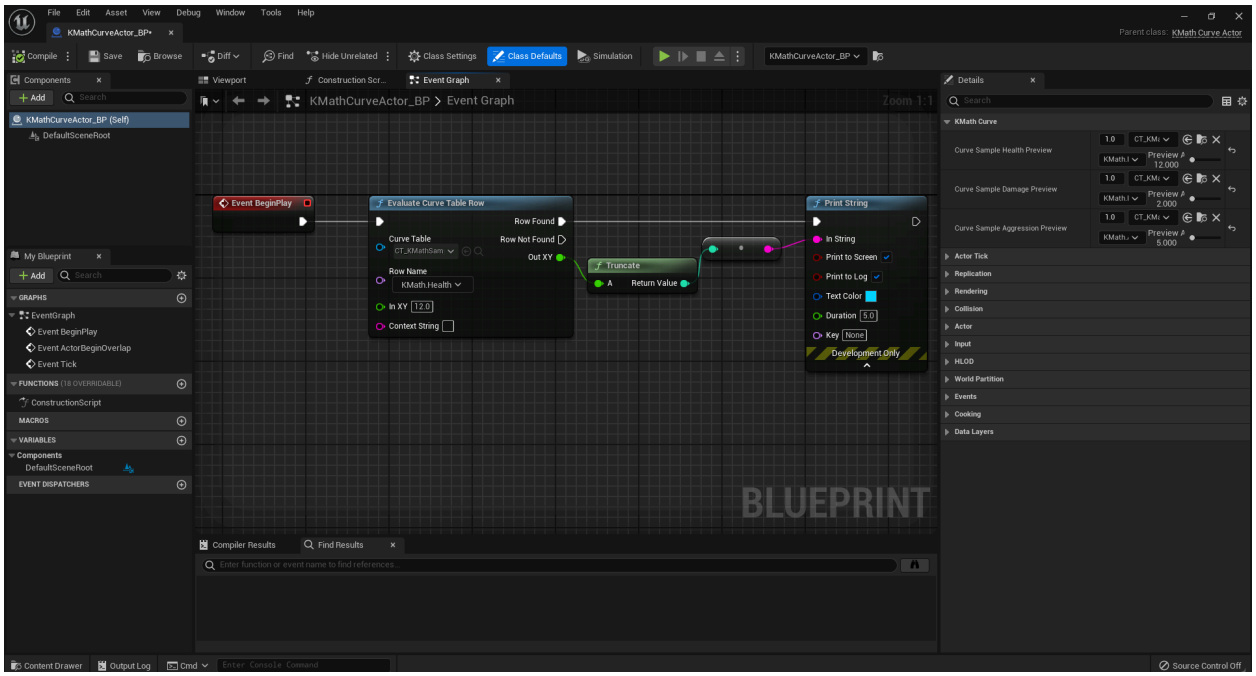
This way, you can script up Blueprints that quickly enable you to preview curve queries in-game while also enabling you to preview the curves using a timeline tool.



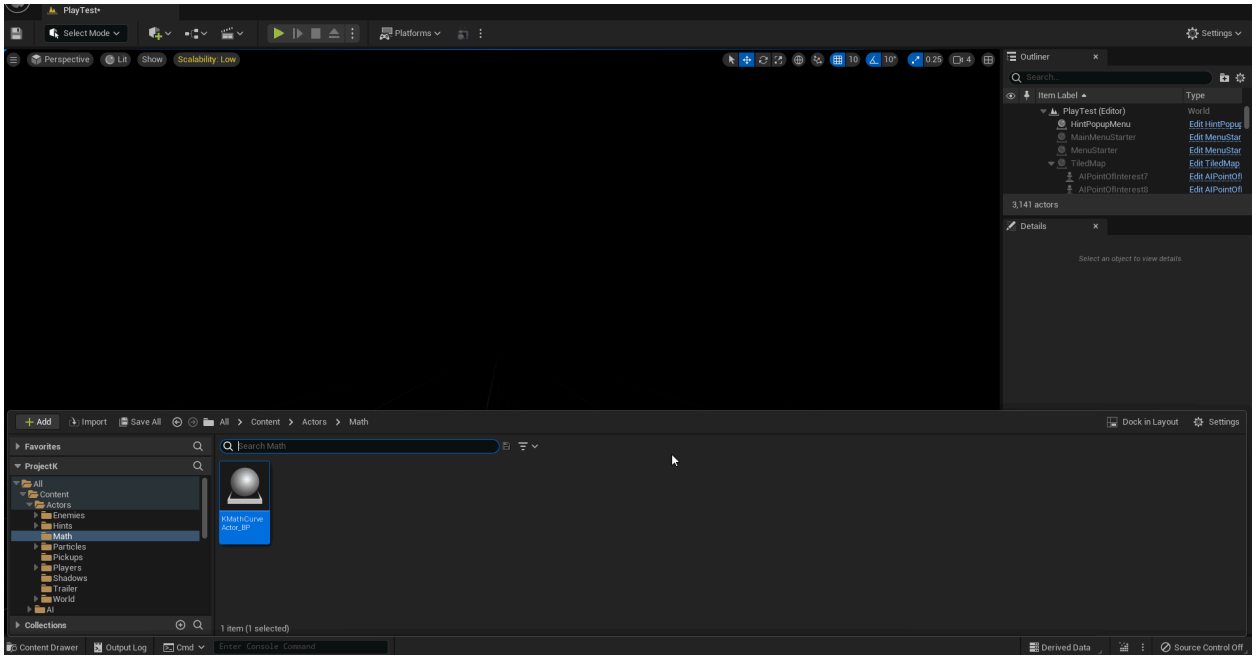
Or you can open the Curve Table itself and inspect the actual curves, which you can see under CurveTable'/Game/Math/CurveTables/CT_KMathSample.CT_KMathSample'



Or you can script up Blueprints that enable you to drop the KMath Actor in game to query different values depending on what you want to test, like the following quick script.



Which debug prints a flush to the game as a quick test like the following.



At the end of the day, KCurveCommander is a system implemented in a way that it should be easy to use and test in a variety of ways.