```
// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.
#include "EnginePrivate.h"
#include "CsvParser.h"
void FKeyHandleMap::Add( const FKeyHandle& InHandle, int32 InIndex )
{
KeyHandlesToIndices.Add( InHandle, InIndex );
}
void FKeyHandleMap::Empty()
{
KeyHandlesToIndices.Empty();
}
void FKeyHandleMap::Remove( const FKeyHandle& InHandle )
{
KeyHandlesToIndices.Remove( InHandle );
}
const int32* FKeyHandleMap::Find( const FKeyHandle& InHandle ) const
{
return KeyHandlesToIndices.Find( InHandle );
}
const FKeyHandle* FKeyHandleMap::FindKey( int32 KeyIndex ) const
{
return KeyHandlesToIndices.FindKey( KeyIndex );
}
int32 FKeyHandleMap::Num() const
{
return KeyHandlesToIndices.Num();
}
TMap<FKeyHandle, int32>::TConstIterator FKeyHandleMap::CreateConstIterator() const
{
return KeyHandlesToIndices.CreateConstIterator();
}
TMap<FKeyHandle, int32>::TIterator FKeyHandleMap::CreateIterator()
{
return KeyHandlesToIndices.CreateIterator();
}
```

```
bool FKeyHandleMap::Serialize(FArchive& Ar)
{
if( Ar.IsTransacting() )
{
// Only allow this map to be saved to the transaction buffer
Ar << KeyHandlesToIndices;
}
return true;
}
bool FKeyHandleMap::operator==(const FKeyHandleMap& Other) const
{
if (KeyHandlesToIndices.Num() != Other.KeyHandlesToIndices.Num())
{
return false;
}
for (TMap<FKeyHandle, int32>::TConstIterator It(KeyHandlesToIndices); It; ++It)
{
int32 const* OtherVal = Other.KeyHandlesToIndices.Find(It.Key());
if (!OtherVal || *OtherVal != It.Value() )
{
return false;
}
}
return true;
}
bool FKeyHandleMap::operator!=(const FKeyHandleMap& Other) const
{
return !(*this==Other);
}
////////////////////////////////////////////////////////////////////////////
// FIndexedCurve
int32 FIndexedCurve::GetIndexSafe(FKeyHandle KeyHandle) const
{
return IsKeyHandleValid(KeyHandle) ? *KeyHandlesToIndices.Find(KeyHandle) :
INDEX_NONE;
}
int32 FIndexedCurve::GetIndex(FKeyHandle KeyHandle) const
{
```

```
return *KeyHandlesToIndices.Find(KeyHandle);
}
FKeyHandle FIndexedCurve::GetKeyHandle(int32 KeyIndex) const
{
check(KeyIndex >= 0 && KeyIndex < GetNumKeys());
EnsureIndexHasAHandle(KeyIndex);
return *KeyHandlesToIndices.FindKey(KeyIndex);
}
TMap<FKeyHandle, int32>::TConstIterator FIndexedCurve::GetKeyHandleIterator() const
{
EnsureAllIndicesHaveHandles();
return KeyHandlesToIndices.CreateConstIterator();
}
bool FIndexedCurve::IsKeyHandleValid(FKeyHandle KeyHandle) const
{
EnsureAllIndicesHaveHandles();
return KeyHandlesToIndices.Find(KeyHandle) != NULL;
}
void FIndexedCurve::EnsureIndexHasAHandle(int32 KeyIndex) const
{
const FKeyHandle* KeyHandle = KeyHandlesToIndices.FindKey(KeyIndex);
if (!KeyHandle)
{
FKeyHandle OutKeyHandle = FKeyHandle();
KeyHandlesToIndices.Add(OutKeyHandle, KeyIndex);
}
}
void FIndexedCurve::EnsureAllIndicesHaveHandles() const
{
if (KeyHandlesToIndices.Num() != GetNumKeys())
{
for (int32 i = 0; i < GetNumKeys(); ++i)
{
EnsureIndexHasAHandle(i);
}
}
}
////////////////////////////////////////////////////////////////////////////
// FRichCurveKey
```

```
static void SetModesFromLegacy(FRichCurveKey& InKey, EInterpCurveMode InterpMode)
{
InKey.InterpMode = RCIM_Linear;
InKey.TangentWeightMode = RCTWM_WeightedNone;
InKey.TangentMode = RCTM_Auto;
if(InterpMode == CIM_Constant)
{
InKey.InterpMode = RCIM_Constant;
}
else if(InterpMode == CIM_Linear)
{
InKey.InterpMode = RCIM_Linear;
}
else
{
InKey.InterpMode = RCIM_Cubic;
if(InterpMode == CIM_CurveAuto || InterpMode == CIM_CurveAutoClamped)
{
InKey.TangentMode = RCTM_Auto;
}
else if(InterpMode == CIM_CurveBreak)
{
InKey.TangentMode = RCTM_Break;
}
else if(InterpMode == CIM_CurveUser)
{
InKey.TangentMode = RCTM_User;
}
}
}
FRichCurveKey::FRichCurveKey(const FInterpCurvePoint<float>& InPoint)
{
SetModesFromLegacy(*this, InPoint.InterpMode);
Time = InPoint.InVal;
Value = InPoint.OutVal;
ArriveTangent = InPoint.ArriveTangent;
ArriveTangentWeight = 0.f;
LeaveTangent = InPoint.LeaveTangent;
LeaveTangentWeight = 0.f;
```

```
}
FRichCurveKey::FRichCurveKey(const FInterpCurvePoint<FVector>& InPoint, int32
ComponentIndex)
{
SetModesFromLegacy(*this, InPoint.InterpMode);
Time = InPoint.InVal;
if(ComponentIndex == 0)
{
Value = InPoint.OutVal.X;
ArriveTangent = InPoint.ArriveTangent.X;
LeaveTangent = InPoint.LeaveTangent.X;
}
else if(ComponentIndex == 1)
{
Value = InPoint.OutVal.Y;
ArriveTangent = InPoint.ArriveTangent.Y;
LeaveTangent = InPoint.LeaveTangent.Y;
}
else
{
Value = InPoint.OutVal.Z;
ArriveTangent = InPoint.ArriveTangent.Z;
LeaveTangent = InPoint.LeaveTangent.Z;
}
ArriveTangentWeight = 0.f;
LeaveTangentWeight = 0.f;
}
bool FRichCurveKey::Serialize(FArchive& Ar)
{
if( Ar.UE4Ver() >= VER_UE4_SERIALIZE_RICH_CURVE_KEY )
{
// Serialization is handled manually to avoid the extra size overhead of UProperty tagging.
// Otherwise with many keys in a rich curve the size can become quite large.
Ar << InterpMode;
Ar << TangentMode;
Ar << TangentWeightMode;
Ar << Time;
Ar << Value;
Ar << ArriveTangent;
```

```
Ar << ArriveTangentWeight;
Ar << LeaveTangent;
Ar << LeaveTangentWeight;
return true;
}
else
{
return false;
}
}
bool FRichCurveKey::operator==( const FRichCurveKey& Curve ) const
{
return (Time == Curve.Time) && (Value == Curve.Value) && (InterpMode ==
Curve.InterpMode) &&
(TangentMode == Curve.TangentMode) && (TangentWeightMode ==
Curve.TangentWeightMode) &&
((InterpMode != RCIM_Cubic) || //also verify if it is cubic that tangents are the same
((ArriveTangent == Curve.ArriveTangent) && (LeaveTangent == Curve.LeaveTangent) ));
}
bool FRichCurveKey::operator!=(const FRichCurveKey& Other) const
{
return !(*this == Other);
}
//////////////////////////////////////////////////////////////////////////
// FRichCurve
FKeyHandle::FKeyHandle()
{
static uint32 LastKeyHandleIndex = 1;
check(LastKeyHandleIndex != 0); // check in the unlikely event that this overflows
Index = ++LastKeyHandleIndex;
}
TArray<FRichCurveKey> FRichCurve::GetCopyOfKeys() const
{
return Keys;
}
TArray<FRichCurveKey>::TConstIterator FRichCurve::GetKeyIterator() const
{
return Keys.CreateConstIterator();
}
```

```
FRichCurveKey& FRichCurve::GetKey(FKeyHandle KeyHandle)
{
EnsureAllIndicesHaveHandles();
return Keys[GetIndex(KeyHandle)];
}
FRichCurveKey FRichCurve::GetKey(FKeyHandle KeyHandle) const
{
EnsureAllIndicesHaveHandles();
return Keys[GetIndex(KeyHandle)];
}
FRichCurveKey FRichCurve::GetFirstKey() const
{
check(Keys.Num() > 0);
return Keys[0];
}
FRichCurveKey FRichCurve::GetLastKey() const
{
check(Keys.Num() > 0);
return Keys[Keys.Num()-1];
}
int32 FRichCurve::GetNumKeys() const
{
return Keys.Num();
}
bool FRichCurve::IsKeyHandleValid(FKeyHandle KeyHandle) const
{
bool bValid = false;
if (FIndexedCurve::IsKeyHandleValid(KeyHandle))
{
bValid = Keys.IsValidIndex( GetIndex(KeyHandle) );
}
return bValid;
}
FKeyHandle FRichCurve::AddKey( const float InTime, const float InValue, const bool
bUnwindRotation, FKeyHandle NewHandle )
{
int32 Index = 0;
for(; Index < Keys.Num() && Keys[Index].Time < InTime; ++Index);
Keys.Insert(FRichCurveKey(InTime, InValue), Index);
```

```
// If we were asked to treat this curve as a rotation value and to unwindow the rotation, then
// we'll look at the previous key and modify the key's value to use a rotation angle that is
// continuous with the previous key while retaining the exact same rotation angle, if at all
necessary
if( Index > 0 && bUnwindRotation )
{
const float OldValue = Keys[ Index - 1 ].Value;
float NewValue = Keys[ Index ].Value;
while( NewValue - OldValue > 180.0f )
{
NewValue -= 360.0f;
}
while( NewValue - OldValue < -180.0f )
{
NewValue += 360.0f;
}
Keys[Index].Value = NewValue;
}

for ( auto It = KeyHandlesToIndices.CreateIterator(); It; ++It)
{
const FKeyHandle& KeyHandle = It.Key();
int32& KeyIndex = It.Value();
if (KeyIndex >= Index) {++KeyIndex;}
}
KeyHandlesToIndices.Add(NewHandle, Index);
return GetKeyHandle(Index);
}
void FRichCurve::DeleteKey(FKeyHandle InKeyHandle)
{
int32 Index = GetIndex(InKeyHandle);

Keys.RemoveAt(Index);
AutoSetTangents();
KeyHandlesToIndices.Remove(InKeyHandle);
for (auto It = KeyHandlesToIndices.CreateIterator(); It; ++It)
{
const FKeyHandle& KeyHandle = It.Key();
int32& KeyIndex = It.Value();
```

```
if (KeyIndex >= Index) {--KeyIndex;}
}
}
FKeyHandle FRichCurve::UpdateOrAddKey(float InTime, float InValue)
{
// Search for a key that already exists at the time and if found, update its value
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
float KeyTime = Keys[KeyIndex].Time;
if( KeyTime > InTime )
{
// All the rest of the keys exist after the key we want to add
// so there is no point in searching
break;
}
if( FMath::IsNearlyEqual( KeyTime, InTime ) )
{
Keys[KeyIndex].Value = InValue;
return GetKeyHandle(KeyIndex);
}
}
// A key wasnt found, add it now
return AddKey( InTime, InValue );
}
FKeyHandle FRichCurve::SetKeyTime( FKeyHandle KeyHandle, float NewTime )
{
if (!IsKeyHandleValid(KeyHandle)) {return KeyHandle;}
FRichCurveKey OldKey = GetKey(KeyHandle);

DeleteKey(KeyHandle);
AddKey(NewTime, OldKey.Value, false, KeyHandle);
// Copy all properties from old key, but then fix time to be the new time
GetKey(KeyHandle) = OldKey;
GetKey(KeyHandle).Time = NewTime;
return KeyHandle;
}
float FRichCurve::GetKeyTime(FKeyHandle KeyHandle) const
{
if (!IsKeyHandleValid(KeyHandle)) {return 0.f;}
```

```
return GetKey(KeyHandle).Time;
}
FKeyHandle FRichCurve::FindKey( float KeyTime ) const
{
int32 Start = 0;
int32 End = Keys.Num()-1;
// Binary search since the keys are in sorted order
while( Start <= End )
{
int32 TestPos = Start + (End-Start) / 2;
float TestKeyTime = Keys[TestPos].Time;
if( TestKeyTime == KeyTime )
{
return GetKeyHandle( TestPos );
}
else if( TestKeyTime < KeyTime )
{
Start = TestPos+1;
}
else
{
End = TestPos-1;
}
}
return FKeyHandle();
}
void FRichCurve::SetKeyValue(FKeyHandle KeyHandle, float NewValue, bool
bAutoSetTangents)
{
if (!IsKeyHandleValid(KeyHandle)) {return;}
GetKey(KeyHandle).Value = NewValue;
if ( bAutoSetTangents )
{
AutoSetTangents();
}
}
float FRichCurve::GetKeyValue(FKeyHandle KeyHandle) const
{
if (!IsKeyHandleValid(KeyHandle)) {return 0.f;}
```

```
return GetKey(KeyHandle).Value;
}
void FRichCurve::ShiftCurve(float DeltaTime)
{
// Indices will not change, so we are ok to do this without regenerating the index handle map
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
Keys[KeyIndex].Time += DeltaTime;
}
}
void FRichCurve::ScaleCurve(float ScaleOrigin, float ScaleFactor)
{
// Indices will not change, so we are ok to do this without regenerating the index handle map
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
Keys[KeyIndex].Time = (Keys[KeyIndex].Time - ScaleOrigin) * ScaleFactor + ScaleOrigin;
}
}
void FRichCurve::SetKeyInterpMode(FKeyHandle KeyHandle, ERichCurveInterpMode
NewInterpMode)
{
if (!IsKeyHandleValid(KeyHandle)) {return;}
GetKey(KeyHandle).InterpMode = NewInterpMode;
AutoSetTangents();
}
void FRichCurve::SetKeyTangentMode(FKeyHandle KeyHandle, ERichCurveTangentMode
NewTangentMode)
{
if (!IsKeyHandleValid(KeyHandle)) {return;}
GetKey(KeyHandle).TangentMode = NewTangentMode;
AutoSetTangents();
}
void FRichCurve::SetKeyTangentWeightMode(FKeyHandle KeyHandle,
ERichCurveTangentWeightMode NewTangentWeightMode)
{
if (!IsKeyHandleValid(KeyHandle)) {return;}
GetKey(KeyHandle).TangentWeightMode = NewTangentWeightMode;
AutoSetTangents();
}
```

```
ERichCurveInterpMode FRichCurve::GetKeyInterpMode(FKeyHandle KeyHandle) const
{
if (!IsKeyHandleValid(KeyHandle)) {return RCIM_Linear;}
return GetKey(KeyHandle).InterpMode;
}
ERichCurveTangentMode FRichCurve::GetKeyTangentMode(FKeyHandle KeyHandle) const
{
if (!IsKeyHandleValid(KeyHandle)) { return RCTM_Auto; }
return GetKey(KeyHandle).TangentMode;
}
void FRichCurve::GetTimeRange(float& MinTime, float& MaxTime) const
{
if(Keys.Num() == 0)
{
MinTime = 0.f;
MaxTime = 0.f;
}
else
{
MinTime = Keys[0].Time;
MaxTime = Keys[Keys.Num()-1].Time;
}
}
/* Finds min/max for cubic curves:
Looks for feature points in the signal(determined by change in direction of local tangent), these locations are then re-examined in closer detail recursively */
template<class T>
void FeaturePointMethod(T& Function , float StartTime, float EndTime, float StartValue,float Mu, int Depth, int MaxDepth, float& MaxV, float& MinVal)
{
if(Depth < MaxDepth)
{
float PrevValue = StartValue;
float PrevTangent = StartValue - Function.Eval(StartTime-Mu);
EndTime += Mu;
for(float f = StartTime + Mu;f < EndTime; f += Mu)
{
float Value = Function.Eval(f);
```

```
MaxV = FMath::Max(Value, MaxV);
MinVal = FMath::Min(Value, MinVal);
float CurTangent = Value - PrevValue;

//Change direction? Examine this area closer
if(FMath::Sign(CurTangent) != FMath::Sign(PrevTangent))
{
//feature point centered around the previous tangent
float FeaturePointTime = f-Mu*2.0f;
FeaturePointMethod(Function, FeaturePointTime, f, Function.Eval(FeaturePointTime),
Mu*0.4f,Depth+1, MaxDepth, MaxV, MinVal);
}
PrevTangent = CurTangent;
PrevValue = Value;
}
}
}
void FRichCurve::GetValueRange(float& MinValue, float& MaxValue) const
{
if(Keys.Num() == 0)
{
MinValue = MaxValue = 0.f;
}
else
{
int32 LastKeyIndex = Keys.Num()-1;
MinValue = MaxValue = Keys[0].Value;
for(int32 i=0; i<Keys.Num(); i++)
{
const FRichCurveKey& Key = Keys[i];

MinValue = FMath::Min(MinValue, Key.Value);
MaxValue = FMath::Max(MaxValue, Key.Value);
if(Key.InterpMode == RCIM_Cubic && i != LastKeyIndex)
{
const FRichCurveKey& NextKey = Keys[i+1];
float TimeStep = (NextKey.Time - Key.Time)*0.2f;
FeaturePointMethod(*this, Key.Time, NextKey.Time, Key.Value, TimeStep, 0, 3, MaxValue,
MinValue );
```

```
}
}
}
}
void FRichCurve::Reset()
{
Keys.Empty();

KeyHandlesToIndices.Empty();
}
void FRichCurve::AutoSetTangents(float Tension)
{
// Iterate over all points in this InterpCurve
for(int32 KeyIndex=0; KeyIndex<Keys.Num(); KeyIndex++)
{
FRichCurveKey& Key = Keys[KeyIndex];
float ArriveTangent = Key.ArriveTangent;
float LeaveTangent = Key.LeaveTangent;
if(KeyIndex == 0)
{
if(KeyIndex < Keys.Num()-1) // Start point
{
// If first section is not a curve, or is a curve and first point has manual tangent setting.
if( Key.TangentMode == RCTM_Auto )
{
LeaveTangent = 0.0f;
}
}
}
else
{

if(KeyIndex < Keys.Num()-1) // Inner point
{
FRichCurveKey& PrevKey = Keys[KeyIndex-1];
if( Key.InterpMode == RCIM_Cubic && (Key.TangentMode == RCTM_Auto ))
{
FRichCurveKey& NextKey = Keys[KeyIndex+1];
ComputeCurveTangent(
```

```
Keys[ KeyIndex - 1 ].Time, // Previous time
Keys[ KeyIndex - 1 ].Value, // Previous point
Keys[ KeyIndex ].Time, // Current time
Keys[ KeyIndex ].Value, // Current point
Keys[ KeyIndex + 1 ].Time, // Next time
Keys[ KeyIndex + 1 ].Value, // Next point
Tension, // Tension
false, // Want clamping?
ArriveTangent ); // Out
// In 'auto' mode, arrive and leave tangents are always the same
LeaveTangent = ArriveTangent;
}
else if( PrevKey.InterpMode == RCIM_Constant || Key.InterpMode == RCIM_Constant )
{
if(Keys[ KeyIndex - 1 ].InterpMode != RCIM_Cubic)
{
ArriveTangent = 0.0f;
}
LeaveTangent = 0.0f;
}

}
else // End point
{
// If last section is not a curve, or is a curve and final point has manual tangent setting.
if( Key.InterpMode == RCIM_Cubic && Key.TangentMode == RCTM_Auto)
{
ArriveTangent = 0.0f;
}
}
}
Key.ArriveTangent = ArriveTangent;
Key.LeaveTangent = LeaveTangent;
}
}
/** Util to find float value on bezier defined by 4 control points */
static float BezierInterp(float P0, float P1, float P2, float P3, float Alpha)
{
const float P01 = FMath::Lerp(P0, P1, Alpha);
```

```
const float P12 = FMath::Lerp(P1, P2, Alpha);
const float P23 = FMath::Lerp(P2, P3, Alpha);
const float P012 = FMath::Lerp(P01, P12, Alpha);
const float P123 = FMath::Lerp(P12, P23, Alpha);
const float P0123 = FMath::Lerp(P012, P123, Alpha);
return P0123;
}
static float BezierInterp2(float P0, float Y1, float Y2, float P3, float mu)
{
float P1 = ( -5.f* P0 + 18.f* Y1 - 9.f *Y2 + 2.f* P3 ) / 6.f;
float P2 = ( 2.f* P0 - 9.f* Y1 + 18.f *Y2 - 5.f* P3 ) / 6.f;
float A = P3 - 3.0f*P2 + 3.0f*P1 - P0;
float B = 3.0f*P2 - 6.0f*P1 + 3.0f*P0;
float C = 3.0f*P1 - 3.0f*P0;
float D = P0;
float Result = A * (mu*mu*mu) + B*(mu*mu) + C * mu + D;
return Result;
}
float FRichCurve::Eval( const float InTime, float DefaultValue ) const
{
const int32 NumKeys = Keys.Num();
// If no keys in curve, return the Default value we passed in.
if( NumKeys == 0 )
{
return DefaultValue;
}
// If only one point, or before the first point in the curve, return the first points value.
if( NumKeys < 2 || (InTime <= Keys[0].Time) )
{
return Keys[0].Value;
}
// If beyond the last point in the curve, return its value.
if( InTime >= Keys[NumKeys-1].Time )
{
return Keys[NumKeys-1].Value;
}
// Somewhere with curve range - linear search to find value.
for( int32 i=1; i<NumKeys; i++ )
{
```

```
if( InTime < Keys[i].Time )
{
const float Diff = Keys[i].Time - Keys[i-1].Time;
if( Diff > 0.f && Keys[i-1].InterpMode != RCIM_Constant )
{
const float Alpha = (InTime - Keys[i-1].Time) / Diff;
const float P0 = Keys[i-1].Value;
const float P3 = Keys[i].Value;
if( Keys[i-1].InterpMode == RCIM_Linear )
{
return FMath::Lerp( P0, P3, Alpha );
}
else
{
const float OneThird = 1.0f/3.0f;
const float P1 = P0 + (Keys[i-1].LeaveTangent * Diff*OneThird);
const float P2 = P3 - (Keys[i].ArriveTangent * Diff*OneThird);
//1st and 2nd derivatives of 0.0
// float QuinticHermite = 6.0f*FMath::Pow(Alpha,5.0f) - 15.0f*FMath::Pow(Alpha,4.0f) +
10.0f*FMath::Pow(Alpha,3.0f);
return BezierInterp( P0, P1, P2, P3, Alpha );
//return FMath::CubicInterp( Keys(i-1).Value, Keys(i-1).LeaveTangent * Diff, Keys(i).Value,
Keys(i).ArriveTangent * Diff, Alpha );
}
}
else
{
return Keys[i-1].Value;
}
}
}
// Shouldn't really reach here.
return Keys[NumKeys-1].Value;
}
bool FRichCurve::operator==( const FRichCurve& Curve ) const
{
if(Keys.Num() != Curve.Keys.Num())
{
return false;
```

```
}
for(int32 i = 0;i<Keys.Num();++i)
{
if(!(Keys[i] == Curve.Keys[i]))
{
return false;
}
}
return true;
}
//////////////////////////////////////////////////////////////////////////
// UCurveBase
UCurveBase::UCurveBase(const FObjectInitializer& ObjectInitializer)
: Super(ObjectInitializer)
{
}
#if WITH_EDITORONLY_DATA
void UCurveBase::GetAssetRegistryTags(TArray<FAssetRegistryTag>& OutTags) const
{
OutTags.Add( FAssetRegistryTag(SourceFileTagName(), ImportPath,
FAssetRegistryTag::TT_Hidden) );
Super::GetAssetRegistryTags(OutTags);
}
#endif
void UCurveBase::GetTimeRange(float& MinTime, float& MaxTime) const
{
TArray<FRichCurveEditInfoConst> Curves = GetCurves();
if(Curves.Num() > 0)
{
check(Curves[0].CurveToEdit);
Curves[0].CurveToEdit->GetTimeRange(MinTime, MaxTime);
for(int32 i=1; i<Curves.Num(); i++)
{
float CurveMin, CurveMax;
check(Curves[i].CurveToEdit != NULL);
Curves[i].CurveToEdit->GetTimeRange(CurveMin, CurveMax);
MinTime = FMath::Min(CurveMin, MinTime);
MaxTime = FMath::Max(CurveMax, MaxTime);
}
```

```
}
}
void UCurveBase::GetValueRange(float& MinValue, float& MaxValue) const
{
TArray<FRichCurveEditInfoConst> Curves = GetCurves();
if(Curves.Num() > 0)
{
check(Curves[0].CurveToEdit);
Curves[0].CurveToEdit->GetValueRange(MinValue, MaxValue);
for(int32 i=1; i<Curves.Num(); i++)
{
float CurveMin, CurveMax;
check(Curves[i].CurveToEdit != NULL);
Curves[i].CurveToEdit->GetValueRange(CurveMin, CurveMax);
MinValue = FMath::Min(CurveMin, MinValue);
MaxValue = FMath::Min(CurveMax, MaxValue);
}
}
}
void UCurveBase::ModifyOwner()
{
Modify(true);
}
void UCurveBase::MakeTransactional()
{
SetFlags(GetFlags() | RF_Transactional);
}
void UCurveBase::OnCurveChanged()
{
}
void UCurveBase::ResetCurve()
{
TArray<FRichCurveEditInfo> Curves = GetCurves();
for(int32 CurveIdx=0; CurveIdx<Curves.Num(); CurveIdx++)
{
if(Curves[CurveIdx].CurveToEdit != NULL)
{
Curves[CurveIdx].CurveToEdit->Reset();
}
```

```
}
}
TArray<FString> UCurveBase::CreateCurveFromCSVString(const FString& InString)
{
// Array used to store problems about curve import
TArray<FString> OutProblems;
TArray<FRichCurveEditInfo> Curves = GetCurves();
const int32 NumCurves = Curves.Num();
const FCsvParser Parser(InString);
const FCsvParser::FRows& Rows = Parser.GetRows();
if(Rows.Num() == 0)
{
OutProblems.Add(FString(TEXT("No data.")));
return OutProblems;
}
// First clear out old data.
ResetCurve();
// Each row represents a point
for(int32 RowIdx=0; RowIdx<Rows.Num(); RowIdx++)
{
const TArray<const TCHAR*>& Cells = Rows[RowIdx];
const int32 NumCells = Cells.Num();
// Need at least two cell, Time and one Value
if(NumCells < 2)
{
OutProblems.Add(FString::Printf(TEXT("Row '%d' has less than 2 cells."), RowIdx));
continue;
}
float Time = FCString::Atof(Cells[0]);
for(int32 CellIdx=1; CellIdx<NumCells && CellIdx<(NumCurves+1); CellIdx++)
{
FRichCurve* Curve = Curves[CellIdx-1].CurveToEdit;
if(Curve != NULL)
{
FKeyHandle KeyHandle = Curve->AddKey(Time, FCString::Atof(Cells[CellIdx]));
Curve->SetKeyInterpMode(KeyHandle, RCIM_Linear);
}
}
```

```
// If we get more cells than curves (+1 for time cell)
if(NumCells > (NumCurves + 1))
{
OutProblems.Add(FString::Printf(TEXT("Row '%d' has too many cells for the curve(s)."),
RowIdx));
}
// If we got too few cells
else if(NumCells < (NumCurves + 1))
{
OutProblems.Add(FString::Printf(TEXT("Row '%d' has too few cells for the curve(s)."),
RowIdx));
}
}
Modify(true);
return OutProblems;
}
//////////////////////////////////////////////////////////////////////////
int32 FIntegralCurve::GetNumKeys() const
{
return Keys.Num();
}
bool FIntegralCurve::IsKeyHandleValid(FKeyHandle KeyHandle) const
{
bool bValid = false;
if (FIndexedCurve::IsKeyHandleValid(KeyHandle))
{
bValid = Keys.IsValidIndex( GetIndex(KeyHandle) );
}
return bValid;
}
int32 FIntegralCurve::Evaluate(float Time) const
{
for (int32 i = 0; i < Keys.Num(); ++i)
{
if (Time < Keys[i].Time)
{
return Keys[FMath::Max(0, i - 1)].Value;
}
}
```

```
return Keys.Num() ? Keys.Last().Value : 0;
}
TArray<FIntegralKey>::TConstIterator FIntegralCurve::GetKeyIterator() const
{
return Keys.CreateConstIterator();
}
FKeyHandle FIntegralCurve::AddKey( float InTime, int32 InValue, FKeyHandle InKeyHandle )
{
int32 Index = 0;
for(; Index < Keys.Num() && Keys[Index].Time < InTime; ++Index);
Keys.Insert(FIntegralKey(InTime, InValue), Index);

for (auto It = KeyHandlesToIndices.CreateIterator(); It; ++It)
{
const FKeyHandle& KeyHandle = It.Key();
int32& KeyIndex = It.Value();
if (KeyIndex >= Index) {++KeyIndex;}
}
KeyHandlesToIndices.Add(InKeyHandle, Index);
return GetKeyHandle(Index);
}
void FIntegralCurve::DeleteKey(FKeyHandle InKeyHandle)
{
int32 Index = GetIndex(InKeyHandle);

Keys.RemoveAt(Index);
KeyHandlesToIndices.Remove(InKeyHandle);
for (auto It = KeyHandlesToIndices.CreateIterator(); It; ++It)
{
const FKeyHandle& KeyHandle = It.Key();
int32& KeyIndex = It.Value();
if (KeyIndex >= Index) {--KeyIndex;}
}
}
FKeyHandle FIntegralCurve::UpdateOrAddKey( float Time, int32 Value )
{
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
float KeyTime = Keys[KeyIndex].Time;
```

```
if( KeyTime > Time )
{
// All the rest of the keys exist after the key we want to add
// so there is no point in searching
break;
}
if( KeyTime == Time )
{
Keys[KeyIndex].Value = Value;
return GetKeyHandle(KeyIndex);
}
}
// A key wasnt found, add it now
return AddKey( Time, Value );
}
FKeyHandle FIntegralCurve::SetKeyTime( FKeyHandle KeyHandle, float NewTime )
{
if (!IsKeyHandleValid(KeyHandle)) {return KeyHandle;}
FIntegralKey OldKey = GetKey(KeyHandle);

DeleteKey(KeyHandle);
AddKey(NewTime, OldKey.Value, KeyHandle);
// Copy all properties from old key, but then fix time to be the new time
GetKey(KeyHandle) = OldKey;
GetKey(KeyHandle).Time = NewTime;
return KeyHandle;
}
float FIntegralCurve::GetKeyTime(FKeyHandle KeyHandle) const
{
if (!IsKeyHandleValid(KeyHandle)) {return 0.f;}
return GetKey(KeyHandle).Time;
}
void FIntegralCurve::ShiftCurve(float DeltaTime)
{
// Indices will not change, so we are ok to do this without regenerating the index handle map
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
Keys[KeyIndex].Time += DeltaTime;
```

```
}
}
void FIntegralCurve::ScaleCurve(float ScaleOrigin, float ScaleFactor)
{
// Indices will not change, so we are ok to do this without regenerating the index handle map
for( int32 KeyIndex = 0; KeyIndex < Keys.Num(); ++KeyIndex )
{
Keys[KeyIndex].Time = (Keys[KeyIndex].Time - ScaleOrigin) * ScaleFactor + ScaleOrigin;
}
}
FIntegralKey& FIntegralCurve::GetKey(FKeyHandle KeyHandle)
{
EnsureAllIndicesHaveHandles();
return Keys[GetIndex(KeyHandle)];
}
FIntegralKey FIntegralCurve::GetKey(FKeyHandle KeyHandle) const
{
EnsureAllIndicesHaveHandles();
return Keys[GetIndex(KeyHandle)];
}
```