Large World Coordinates Project Conversion Guidelines

This document serves as a guide to convert your UE4 Projects to UE5 with minimal precision loss.

Learn to use this **Beta** feature, but use caution when shipping with it.

Large World Coordinates (**LWC**) is designed to slot into your existing project seamlessly. You can import your project into **Unreal Engine 5** and follow this guide for conversion, however, it is possible that a limited set of issues may arise. In source code, variant types have a typedef alias that will default to the double type which matches the original **Unreal Engine 4**(UE4) name. Your existing code will be recompiled to support doubles automatically.

Modified Aliased Types

For compatibility in the future, we recommend you use the FVector alias wherever possible, only falling back to the explicit FVector3f or FVector3d types where you absolutely need to ensure a particular type.

The table below displays the following types that have been converted to variants:

Default (Double Variant Alias)	Float Variant	Double Variant
FVector	FVector3f	FVector3d
FVector2D	FVector2f	FVector2d
FVector4	FVector4f	FVector4d
FMatrix (FScaleMatrix,FQuatRotationTranslationMatrix, and more.)	FMatrix44f	FMatrix44d
FPlane	FPlane4f	FPlane4d
FQuat	FQuat4f	FQuat4d
FRotator	FRotator3f	FRotator3d
FTransform	FTransform3f	FTransform3d
FBox	FBox3f	FBox3d
FBox2D	FBox2f	FBox2d
FSphere	FSphere3f	FSphere3d
FBoxSphereBounds	FBoxSphereBounds3f	FBoxSphereBounds3d
FCapsuleShape	FCapsuleShape3f	FCapsuleShape3d
FDualQuat	FDualQuat4f	FDualQuat4d
FRay	FRay3f	FRay3d

To maintain compatibility in the future we do not recommend using the variant types unless absolutely necessary. Instead, you should use the original versions listed in the alias above, and only use the explicit float and double types in instances where you need to ensure a particular type.

If you want to cast between variants, then you must do this explicitly. For example:

```
FVector Vec(1.0, 2.0, 3.0);

FVector3f AsFloat = FVector3f(Vec);

FVector3f AsFloat = Vec; // Will fail to compile.

```

Copy full snippet

FMath Support for Doubles

FMath functionality has been extended to support doubles. This includes support for standard mathematical operations on the new types.

This includes vectorization support from the VectorRegister located in the UnrealMathSSE.h/UnrealMathNeon.h directory which adds VectorRegister4f and VectorRegister4d types.

Experimenting with Large Worlds

As a result of the beta status of Large Worlds in UE5, the default `WORLD_MAX` size has been left at the UE4 `WORLD_MAX` size of 21km and the engine check on the world bounds remains enabled. There are two options for experimenting with the scale of your large worlds:

You can disable the bounds checks by accessing your WorldSettings class and setting your `bEnableLargeWorlds` boolean to `true`:

```
AWorldSettings::bEnableLargeWorlds = true

```

Copy full snippet

This will keep the value of `WORLD_MAX` at approximately 21 km and provides improved stability for experimentation in the initial release of Unreal Engine 5.0.

Alternatively, you can set the global value of `UE_USE_UE4_WORLD_MAX` to enable larger world bounds:

```
UE_USE_UE4_WORLD_MAX=0

```

Copy full snippet

This will set the `WORLD_MAX` value to approximately 88 million km.

This value may change before future releases of Unreal Engine and may exhibit stability issues that will continually be optimized throughout the development of Unreal Engine 5.

Code Compile Errors

Below you will find several categories of compilation errors and their suggested solutions.

Forward Declaration of Variant Types

When writing forward declaration statements you should use the macro UE_DECLARE_LWC_TYPE.

For example instead of:

```
struct FVector;

```

Copy full snippet

FVector should be declared as:

```
UE_DECLARE_LWC_TYPE (Vector, 3);

```

Copy full snippet

Please refer to `Engine/Source/Runtime/Core/Public/CoreFwd.h` for examples of proper `UE_DECLARE_LWC_TYPE` usage per type.

Forward declaration of variant types is unnecessary for any file that includes a generated.h, or CoreMinimal.h.

Error: C2027/C2371

If your project is generating C2027/C2371 errors for an FVector or other variant type, then a forward declaration of that type as a struct within your code is the most likely cause.

Warnings: "Arguments cause function resolution ambiguity"

You may observe warnings similar to the one above if you're calling multi-argument FMath functions that contain a mix of variant type, floating-point, or constant arguments.

It is important that you **do not ignore function resolution ambiguity warnings** as they may be indicating an underlying loss of precision occurring in your program.

You can correct precision errors by passing explicit template arguments, casting mismatched types, or modifying constants to match the desired type.

For example:

```
FMath::Max(MyVector.X, double(MyFloat));

```

Copy full snippet

Converting Between Float and Double Variant Types Requires Explicit Casting

Converting between float and double variant types in code requires explicit casting to avoid unintended precision or performance issues.

As an example, you may need to account for this behavior when passing an FVector3f into a function that requires a FVector4. The FVector3f should be cast to the FVector, this enables the implicit cast from an FVector to an FVector4 to complete the conversion.

> Explicit casting applies to similar types in internal systems. For example, a Chaos FVec3 to an FVector3f.

Shader Parameters

We do not support double parameters on the GPU. As such, FVector, FVector2D, FVector4, and FMatrix are no longer supported within SHADER_PARAMETER declarations in native code, and will need to be switched to the float variant of the corresponding type.

Runtime Check Failures

You may encounter runtime check failures such as:

Unexpected element array size in TArray::BulkSerialize

These can be a result of bulk serializing a struct that contains a type that was automatically converted to a double variant. This can be fixed by adding a bForcePerElementSerialization parameter based on the archive version:

```
MyArray.BulkSerialize(Ar, Ar.UEVer() <
EUnrealEngineObjectUE5Version::LARGE_WORLD_COORDINATES);

```

Copy full snippet

We recommend you ensure your archive has an up-to-date version.

Precision Issues

The conversion of LWC type components to a double type can introduce precision issues in your converted UE4 projects, particularly if your code is written with the expectation of these components being floats. We recommend you audit your code for these issues after upgrading.

Enable Unsafe Typecast Warnings

Below you will find several methods available to enable your project to receive unsafe typecast warnings.

By Module

You can add the following to your project's build.cs file:

```
UnsafeTypeCastWarningLevel = WarningLevel.Warning;

```

Copy full snippet

While this may generate a considerable number of warnings in a large codebase, it is invaluable for detecting and fixing situations that may lead to a loss of precision.

Some Engine header files will generate unsafe typecast warnings at present. These will be fixed in a future release.

By Single File or Code Block

The following macros enable toggling of unsafe typecast warnings:

Macro	Description
PRAGMA_FORCE_UNSAFE_TYPECAST_WARNINGS	Unsafe typecasts will generate errors past this point regardless of module settings.
PRAGMA_DISABLE_UNSAFE_TYPECAST_WARNINGS	Unsafe typecasts will be ignored past this point regardless of module settings.
PRAGMA_RESTORE_UNSAFE_TYPECAST_WARNINGS	End marker for FORCE/DISABLE block. Behavior reverts to module settings. The CheckBalancedMacros automation script will fail if blocks are not closed correctly.

Ensure No Loss of Precision When Storing A Copy of Type Components.

Code which accesses Type Components directly may require some refactoring to avoid precision loss. Precision errors can lead to invalid code execution, for example:

```
const float X = MyVector.X;

// MyVector.X may be more precise than we expect now.
//SMALL_NUMBER == 1e-8 (0.00000001)
```

```
// Double to float conversion can introduce precision error at as few as 6 significant digits.

if(FMath::Abs(X - OtherVector.X) > SMALL_NUMBER)
{
// If MyVector.X and OtherVector.X are identical we wouldn't expect to get in here.
}

```

Copy full snippet

For a precise double value of `MyVector.X`, the difference between the `X` and the `OtherVector.X` delta can be significant enough to cause the code to follow this path.

For a project that plans to expand beyond the older UE4 float limitations of a 10.5KM bounds, this becomes increasingly dangerous as you move further out from the origin:

```
float X = MyVector.X;

X += 0.5f;

MyVector.X = X; // Precision loss.

```

Copy full snippet

If you are planning to make use of large world coordinates in your converted project, then you need to audit your codebase for these precision errors.

LWC types expose an FReal alias to their underlying component type (float or double). Using an FReal in place of the fundamental types will ensure you avoid precision issues, even if these types should change in the future, for example:

```
FVector::FReal ReallyADoubleNow = FMath::Cos(MyVector.X);

```

Copy full snippet

A refined approach would be to update your float types to double across your codebase:

```
double ReallyADoubleNow = FMath::Cos(MyVector.X);

```

Copy full snippet

Code that accesses type components directly may require refactoring to avoid precision loss.

> A converted UE4 project that intends to stay within the float bounds (~10.5KM from the origin) may experience a precision loss, however it may not negatively affect your project.

Bulk Serialization of Variant Types

By default, bulk serialization of arrays of default core types(FVector) is disabled for converted projects. The engine is required to load each vector component as a float, then convert it to a double. When an affected asset is re-saved, it is written as a double, and bulk serialization will function again. if you are bulk serializing structs containing a variant type, then you will need to convert it to use the float variant, or force a per-instance serialization in the `BulkSerialize` call / disable `TCanBulkSerialize` support.

Reduced Memory Consumption of Core Types

Serialized Core types that are marked with Property Specifiers will automatically recognize a switch between the double and float variants, providing you the ability to retroactively reclaim wasted memory by switching to the float variant at any time.

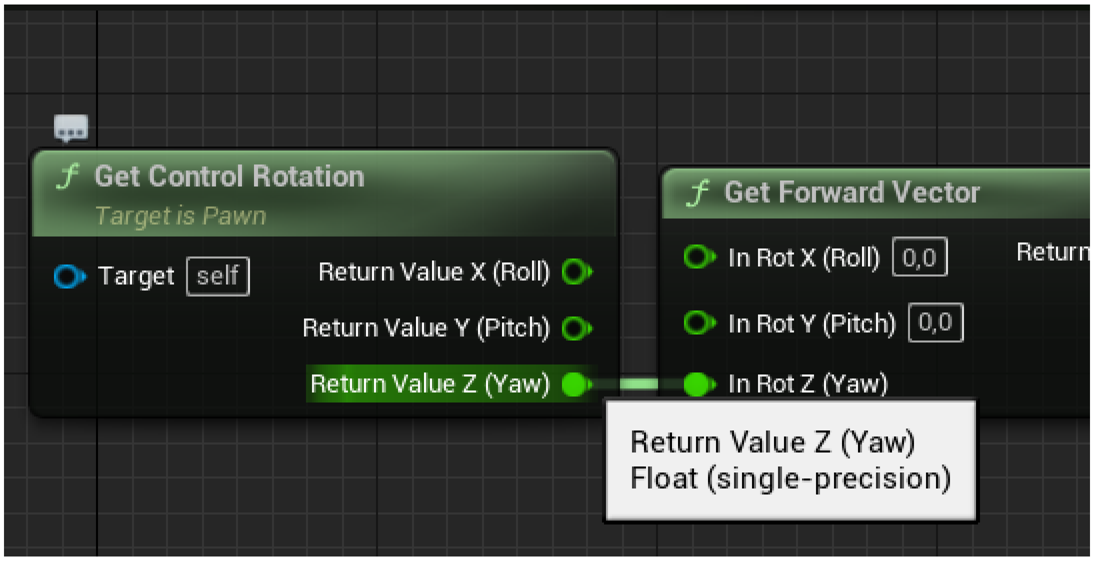
Blueprints

In UE5, all Blueprint Float types are represented by a modified Float type which is capable of working at both single and double precision. This means your Blueprints will automatically support LWC scales. Any float or double types in an imported UE4 blueprint project are promoted to the new type, as a result of removing explicit float / double support in Blueprints.

Source Code Interface

Source code may now expose both float and double types. The **Unreal Header Tool**(**UHT**) will interpret any Blueprint-accessible floating-point type in code as a Blueprint Float with the appropriate single (C++ float) or double (C++ double) precision subtype, enabling automatic conversion of Float values of either precision supplied by any Blueprint node. We recommend caution when using floating-point terminology.

A float in Blueprints can refer to either a single-precision or double-precision float. However, In C++ you need to be explicit about what you want. The term "float" specifically refers to a single precision float, whereas a "double" refers to a double precision float. The Unreal Header Tool recognizes both types as valid for Properties or Function Parameters. If the type is a "float" or "double", Blueprint will treat either as a singular "Float" type, but will indicate whether the float is single or double precision.



From the perspective of a Blueprint user, there's no difference between a single or double precision type, these pins can be connected without any explicit cast nodes. Nonetheless, Unreal Engine may need to perform either a narrowing or widening conversion in these instances. The Blueprint compiler will automatically analyze a Blueprint graph for any potential floating-point conversions. If any are detected, the compiler will automatically insert a cast operation into the underlying code. This works for containers as well.

Developers do not have to do any special fixes with their older Blueprint content to take advantage of Large World Coordinates. By default, Unreal Engine automatically converts float pins to use double precision. If any pins were used for single precision floats in native C++ code, then it ensures that those pins continue to represent single precision floats. This includes native C++ properties, function parameters, and the parameters of any Blueprint functions that are bound to native C++ delegates.

Exposing UFUNCTION Property Specifiers Expecting Float Values

Any method marked with a UFUNCTION Property Specifier that contains a float data value risks introducing imprecision, because the Blueprint Float value is cast to the lower precision float. It is important to audit any existing UFUNCTION property. This helps to determine if switching the parameters or return values to double may be necessary to avoid precision issues in the future. Switching between float and double types is safe to do at any time, and in either direction.

This applies to any K2 nodes that you may have constructed or exposed in code.

Plugins - Guidelines for Published Plugin Authors

When converting existing plugins to support Large Worlds, we advise you to be cautious to ensure your plugin avoids precision loss. We recommend using the general guidelines listed in the Precision Issues section to update your plugin core-type components.

Full World Space Plugins

Certain plugins may require conversion to full world space. This can be done by converting your underlying types to use doubles, although the engine will accommodate the majority of this process when you convert your plugin to UE5, you will need to ensure your code is not creating precision errors by storing results as floats.

For example: FVectors used to represent world coordinates in UE4 will already have been upgraded to use double precision types in UE5, however, if your plugin accesses core types at the component level, then some refactoring of your code may be required to ensure precision is maintained.

World Space Origin Only Plugins

Some plugins only need to work within a local float scale space, and can rely on the engine to render the results at a world space origin. In this case you may want to explore converting the core types used internally by your plugin to float variants to reclaim memory lost to doubles.

Supporting both UE4 and UE5

Plugins that target both engine versions need to be distributed as code to be compiled at source, or as separate binary packages. UE4 does not support core type variants, so you either need to ensure your UE5-compatible code uses only the default core types (compiling as floats in UE4, and as doubles in UE5), or provide separate source code for UE4 and UE5 versions of the plugin.

To ensure the correct precision for intermediate calculations using core type components, they can be handled using either the auto c++ declaration, or doubles.

LWC Rendering Overview

New HLSL types were introduced along with Large World Coordinates and can be found in the LargeWorldCoordinates.ush file. Refer to the LWC Rendering Doc for additional information on how to convert your shader code to UE5.