Large World Coordinates Rendering Overview.

An overview of Rendering Large World Coordinates.

Large World Coordinates (LWCs) uses a math library named **DoubleFloat** that emulates double-precision floating-point arithmetic using pairs of single-precision floating-point numbers. The C++/HLSL types and operators for this library are located in the `Math/DoubleFloat.h` and `DoubleFloat.ush` source files.

The following HLSL types are available:

HLSL Type	Notes
`FDFScalar` `FDFVector2/3/4`	These types are the higher precision counterparts to float1/2/3/4. Internally composed of two floatN vectors: High and Low.
`FDFMatrix`	Similar to float4×4, however, includes an extra float3 PreTranslation coordinate. When multiplying a vector with this type, the vector is first translated with PreTranslation before the float4×4 matrix is applied. This type is useful for transforming to world space (LocalToWorld).

HLSL Type	Notes
`FDFInverseMatrix`	Similar to float4×4, however, includes an extra float3 PostTranslation coordinate. When multiplying a vector with this type, the vector is translated with PostTranslation after the float4×4 matrix is applied. This type is useful for transforming to world space (LocalToWorld).

Operators are prefixed with “DF” and come with several variants that offer a trade-off between precision and performance. The variants are marked with a prefix or postfix in the function name. For example:

- `DFFast*` is a postfix. For example, `DFFastAdd` is a postfix variant. This function is faster, but less precise. Use this variant to get the maximum speed at the cost of precision. For the base operators, each function is documented with a reference to a paper that gives details about its precision bounds.
- `DF*Demote` is a prefix. For example, DFFastAddDemote is a prefix variant. This variant returns a 32-bit result instead of double precision. A `DF*Demote` function is available and is similar to the `LWCToFloat` function. Using `DF*Demote` shaves off unnecessary computations and is more efficient. It is the preferred method of truncation.

> Many of the math functions in the DoubleFloat math library have a substantial performance cost. For guidance on achieving optimal performance and precision, see the Translated World Space section of this page.

FLWCVector and Similar Types

In Unreal Engine, there are several composite types that use a different underlying mathematical type structure designed for use with large world coordinates. These types are prefixed with “FLWC” and can be found in the `LargeWorldCoordinates.ush` source file. Most shaders now use the DoubleFloat types. However, the FLWC types are still used by some systems.

The following HLSL types are available:

HLSL Type	Notes
`FLWCScalar` `FLWCVector2/3/4`	These types are similar to float1/2/3/4, however, they include an additional Tile coordinate. Therefore, a FLWCVector2 is made from a float2 Tile and a float2 Offset. The value represented is calculated by the formula: Tile * TileSize + Offset, where TileSize represents a constant value that is defined as 256k.
`FLWCMatrix` `FLWCInverseMatrix`	These types are similar to float4×4, however, they both include an additional float3 Tile coordinate.
`FLWCMatrix`	Adds the Tile coordinate to the result after multiplying the matrix.
`FLWCInverseMatrix`	Adds the Tile coordinate before multiplying the matrix.
`FLWCMatrix`	This type is useful for transforming to world space (LocalToWorld).
`FLWCInverseMatrix`	This type is useful for transforming from world space (WorldToLocal).

Operations, such as `LWCAdd` or `LWCRsqrt` can be performed on these types. Operations that take LWC input values return LWC outputs (`LWCAdd`), while others return regular float outputs (`LWCRsqrt`). Operations that make coordinates smaller provide you with a return to regular floats.

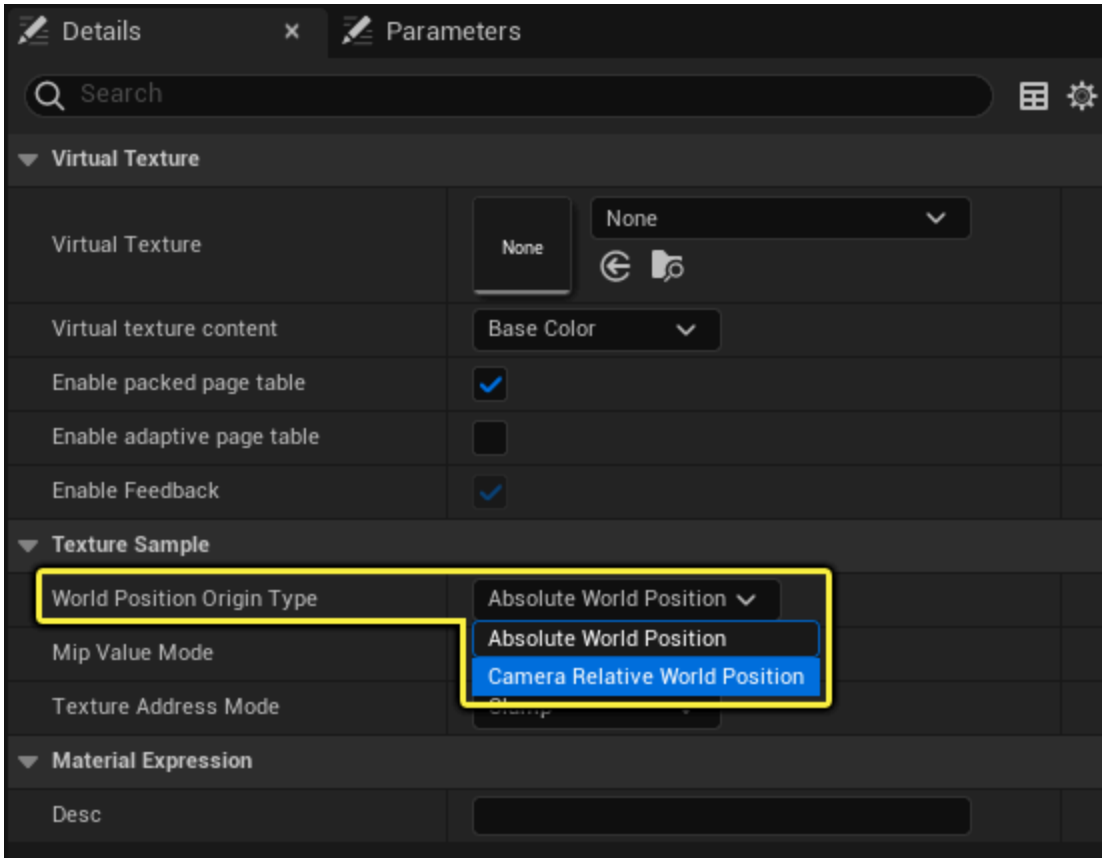
The FLWC types provide inferior precision compared to the DoubleFloat library. For vectors that approach the tile edge, the fixed tile-size means that bits in the **Tile** component go unused while the **Offset** can only store a truncated coordinate. Prior to Unreal Engine 5.4, this resulted in oscillating precision as objects moved through the world, mostly visible as object jittering and vibration. The DoubleFloat math types solve this by removing the fixed size since both components always store as much precision as they can, regardless of the value's magnitude. This is conceptually similar to the difference between fixed-point and floating-point arithmetic.

Materials

The material translator works with LWC values. Some nodes in your materials output LWC values instead of floats. Notably, coordinate-based nodes for world space position, such as WorldPosition, Object, Camera, Actor, and Particle, along with the TransformPosition node output LWC values. Also, most math nodes output LWC types if the input is an LWC type variable.

> There is an added performance cost that comes with using LWC in materials. Absolute worldspace works at LWC scale but costs performance. Math with LWC takes longer to compute, thus is more expensive.

You can set the **World Position Origin Type** property — on most material nodes that take or produce a world space value — to use **Camera Relative World Position** (or translated world space) and achieve optimal precision and performance.



Even though material graphs continue to use the tile-offset system due to performance constraints, you can still see precision improvements when using translated worldspace because backend systems, such as vertex factories, have been updated to use DoubleFloat.

Additional things to keep in mind when using LWC with materials:

- Absolute worldspace works with LWC scale but costs performance.
- Math with LWC takes longer to compute, thus is more expensive.
- Prefixed alias types with “FWS” are found in the `WorldSpaceMath.ush` header. These are used as an abstraction and are either implemented with the `FLWC*` or the `FDF` types. Use these when writing HLSL code that interacts with material graphs, for example, the material template header.

Conversion Guide from UE4 to UE5

If you are migrating your project from Unreal Engine 4 or earlier, you can port your existing HLSL codebase into Unreal Engine 5. Below are some suggestions to ensure a successful conversion.

- Change **View** to **PrimaryView** when accessing values/matrices in the world space (LocalToWorld, PreViewTranslation).
- If your code performing in world space doesn’t compile (for example, it is missing function overloads or type conversions), and if you don’t have the resources to refactor your codebase, you can use the **LWCHackToFloat** function where needed. For example, `LWCHackToFloat(PrimaryView.PreViewTranslation)`.

In the event your code still does not compile:

- Consider changing the shader code to operate in TranslatedWorldSpace instead of WorldSpace.
- World space values require LWC types (`FLWSVector3` instead of float3) and corresponding LWC functions.

LWC types are not included directly inside uniform buffers. The variable is usually stored as its based component (such as Tile and Offset) if an FLWC* type is used. Frequently, entities contain several pieces of data that can all use the same tile coordinate (for example, WorldToLocal, LocalToWorld, and AbsoluteWorldPosition). Therefore, the typical pattern is then to store offset values in the uniform buffer (such as, RelativeWorldToLocal, LocalToRelativeWorld, and RelativeWorldPosition) along with a single shared float3 TilePosition value. This uniform data is “unpacked” into a new struct type that includes the actual LWC values in which many values are initialized with the single TilePosition inside the shader.

There is no Tile coordinates that can be shared for `FDF*` types. It is possible to store a float3 reference point and several non-LWC values that are relative to this point if memory cost or bandwidth is a concern. These can be recombined to DoubleFloat values in the shader using `DFTwoSum(Offset, Base)`. Then you can use the DFFastTwoSum(Offset, Base) (which has half the number of operations) if you can guarantee that |Offset| is greater than |Base|.

Many variables have been switched to LWC variants within the shader code. These are some notable examples:

- `SceneData.ush`, `FPrimitiveSceneData`, and `FInstanceSceneData` have various updated matrices and position vectors.
- Light positions and reflection capture positions.
- Global camera uniforms have various matrices and offsets changed (such as, PreViewTranslation). For example, in SceneView and FNaniteView.

Prior to Unreal Engine 5.0, global camera uniforms were either accessed through `ResolvedView` when rendering meshes (which handles instanced stereo rendering), or `View` when rendering post-processing and other global passes. `View` refers directly to the global uniform buffer in these versions of Unreal Engine. `PrimaryView` has been added as a new alias which refers to the unpacked global view struct. Therefore, names such as `View.PreViewTranslation` have changed to `PrimaryView.PreViewTranslation`. `ResolvedView.PreViewTranslation` continues to function where applicable and is valid to access non-LWC quantities from the regular View alias.

Switching these global types provides a general indication of where your shader code needs to be refactored to support LWC. For example:

```
1 float3 WorldPosition = mul(Input.LocalPosition, View.LocalToWorld);
2
3 //The above line no longer compile, and will need to be refactored to:
4
5 FDFVector3 WorldPosition = DFMultiply(Input.LocalPosition,
  PrimaryView.LocalToWorld);
```

Copy full snippet

The initial LWC pass does not check to ensure that all shaders work correctly with LWC. You can use the global function `LWCHackToFloat` to convert a LWC type to a non-LWC type.

`LWCHackToFloat` behaves as a wrapper around the `DFDemote` and `LWCToFloat` functions. The non-wrapped versions are used when you know a conversion is safe at LWC-scale. `LWCHackToFloat` is a searchable marker. If you didn't have time to refactor your codebase from switching `WorldPosition` from `float3` to `FDFVector3`, you could instead use the following code:

```
float3 WorldPosition = mul(Input.LocalPosition, LWCHackToFloat(PrimaryView.L
```

Copy full snippet

If you are upgrading your project from Unreal Engine 5.0 or later and are using `FLWC*` types, the functions `DFFromTileOffset` and `DFToTileOffset` convert between these types and the new DoubleFloat ones. There's also `DFFromTileOffset_Hack` and `DFToTileOffset_Hack` which are aliases that are used as a marker, similar to `LWCHackToFloat`. Converting between types incurs performance and precision loss, so these should be used only when necessary.

TranslatedWorldSpace is an existing engine concept that is used in Unreal Engine shaders. It was previously intended to increase precision by working relative to the camera origin. However, this behavior is useful for LWC because it is safe to use floats when working in TranslatedWorldSpace. Rather than convert WorldPosition to a double-precision value, we recommend refactoring the function to use translated world space instead, which results in superior performance while retaining high precision. For example:

```
float3 TranslatedWorldPosition = mul(Input.LocalPosition, PrimaryView.LocalT
```

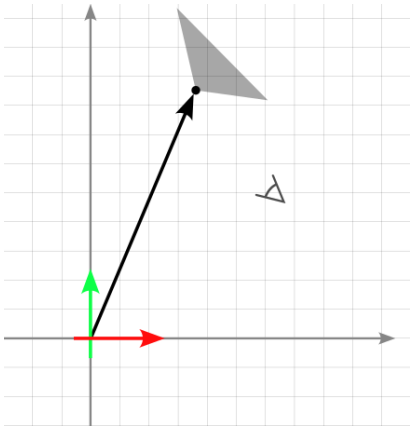
Copy full snippet

Translated World Space

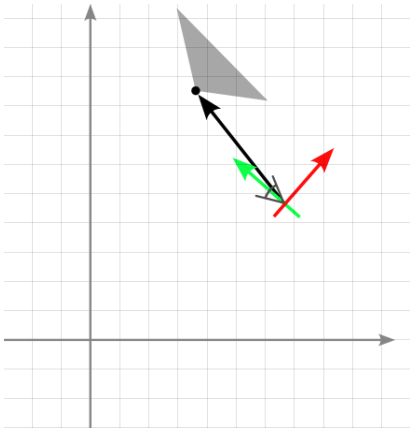
For optimal performance and precision, you can convert from absolute world space (left) to a translated world space (right), such as **Camera Relative World Space**. Ideally, you should

convert to a translated world space as soon as possible in your project to take advantage of the benefits.

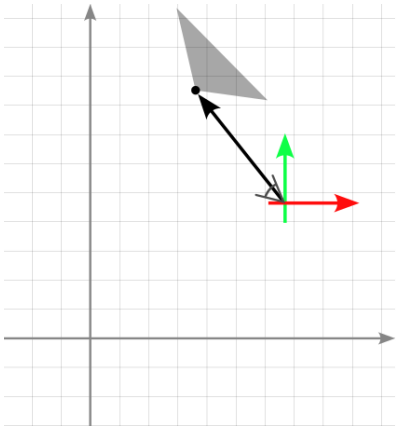
Below shows examples of absolute world space (left), camera space (middle), and translated world space (right).



Absolute World Space



Camera Space



Translated World Space (Camera-relative World Space)

You can convert your worldspace using the function `LWCAdd(WorldPosition, PreViewTranslation)` with the tile-offset types. The fastest method for this in the DoubleFloat library is `DFFastToTranslatedWorld()`. This method can also be used to convert `XToWorld` or `WorldToX` matrices to `XToTranslatedWorld` and `TranslatedWorldToX`. The fastest way to convert a local position to a translated worldspace with a LocalToWorld matrix is to use `DFTransformToTranslatedWorld()`.

> These functions apply specific optimizations to reduce overhead and are faster than using `DFFastAdd`, and other similar functions.

Mathematical Background for DoubleFloat

The `FDFVector` types consist of a High and Low single-precision vector. The High component is equal to the value rounded to the nearest 32-bit float, while the Low value captures (most of) the error that is introduced in rounding.

Conversion from a 64-bit double to a `FDFScalar` is performed like this:

```
FDFScalar(double Input)
{
float High = (float)Input;
float Low = (float)(Input - High);
}
```

Copy full snippet

The table below is an example of various binary representations of the number 1.1 rounded to 17 decimal places:

`(double)1.1`	1.1000000000000001
`(float)1.1`	1.1000000238418579
`FDFScalar.High`	1.1000000238418579
`FDFScalar.Low`	-2.3841858265427618e-08

The following are some resources on this subject:

- For an academic introduction, see the paper Extended-Precision Floating-Point Numbers for GPU Computation by Andrew Thall (2006).
- For a more comprehensive and rigorous resource, see Formalization of Double-Word Arithmetic, and Comments on “Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic” by Jean-Michel Muller and Laurence Rideau (2022).